SAMS

22425

Enhancing Your Apple® II and IIe

Volume 2

Don Lancaster

Enhancing Your Apple® II and IIe

Volume 2

Don Lancaster heads Synergetics, a new-age prototyping and consulting firm involved in limit pushing micro applications and electronic design. He is the well-known author of the classic *CMOS* and *TTL Cookbooks*. He is one of the microcomputer pioneers, having introduced the first hobbyist projects involving integrated circuits, the first affordable digital counting modules, the first low-cost TVT-1 video display terminal, the earliest personal computing keyboards, and lots more. Don's numerous books and articles on personal computing have set new standards as understandable, useful, and exciting technical writing. Don's other interests include ecological studies, fire fighting, cave exploration, bicycling, and tinaja questing.

The DON LANCASTER Library

Assembly Cookbook for the Apple II/IIe 22331
Active Filter Cookbook 21168
Cheap Video Cookbook 21524
CMOS Cookbook 21398
Enhancing Your Apple II, Vol. 1 21822
Hexadecimal Chronicles 21802
Micro Cookbook, Vol. 1 (Fundamentals) 21828
Micro Cookbook, Vol. 2 (Machine Language) 21829
TTL Cookbook 21035
Son of Cheap Video 21723
TV Typewriter Cookbook 21313
The Incredible Secret Money Machine (Available only from Synergetics)

Enhancing Your Apple® II and IIe

Volume 2

by

Don Lancaster

Howard W. Sams & Co., Inc.

4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

FIRST EDITION
FIRST PRINTING — 1985

International Standard Book Number: 0-672-22425-9
Library of Congress Catalog Card Number: 85-50022

Edited by: *Welborn Associates, Inc.*
Illustrated by: *David K. Cripe*

Printed in the United States of America.

Contents

Introduction

Per your feedback card and hotline requests, Volume 2 in this continuing series has far more in the way of brand new, in-depth, and ready-to-use software and hardware concepts. These will push both the real and imagined limits of the Apple II and IIe to the utmost.

Around 75% of you using the response cards have verified receiving your own personal "IT" messages from your own Apple. I guess the main reason for the 25% no-shows is that the Apple involved suspects any no-shows of being users instead of hackers.

As we'll find out in Enhancement 13, the penalty for an Apple revealing "IT" messages to a user is severe indeed. At any rate, the secret "IT" messages continue fast and furious, at least from my own IIe . . .

• • •

Did you know that you can save up to 40% of the cost of all the software you buy? Just buy through a distributor, rather than a retailer. It is that easy.

There usually is a minimum order of $100 or so, and a letterhead or a tax number may sometimes be needed. Both of which get paid for out of the savings on your first purchase. Just bunch your orders together or go in with a few friends.

Suitable distributors advertise regularly in such magazines as *Computer Retailing, Computer Retail News,* and *Computer Dealer.* Two of my favorite distributors are *Ingram* and *First Software,* but there are many more. Many distributors will accept 800 phone orders and ship on VISA® or UPS COD.

• • •

The words "software pirate" have been much bandied about lately. Up to now, though, these words have lacked a precise and scientific definition. Let's nail them down once and for all.

Since it is an observable, immutable, and unarguable scientific fact that there never has been *any* piece of Apple software *ever* that was worth over $25, we will, henceforth and evermore, define a "software pirate" as anyone who charges over $24.99 for an Apple program.

• • •

How do you keep others from stealing code that you have written? Certainly not by copy "protection," for copy "protection" is nothing but a ridiculous game that penalizes and inconveniences the legitimate user while waving a red-flag challenge and providing unbeatable entertainment to all the rest of hackerdom.

To protect yourself: First through ninety-ninth, reduce the final user price of your product.

One-hundredth on your survival list is to have as much added value as possible *outside* of your actual code. Do this with a thorough and professional user manual; with support forms, charts, or game pieces; with a free backup copy or detailed and "up-front" copy instructions and a $5.00 or less exchange policy; with source code easily available; and with personalized and long-term hotline and mail product support.

• • •

Newcomers to Appledom are often confused by the differences between the Apple DOS and CPM operating systems.

The two are very simple to tell apart. Just run the program. If the program is innovative, useful, and creative, then it is running under DOS 3.3 or 3.3e.

• • •

My Apple is continually amazed that the single most important reason for bootleg copies of programs existing is never mentioned in polite company and has not once appeared in print.

It turns out that a bootleg copy is often the *only* way available that you can get prompt, courteous, and well-informed service at a reasonable price. Let's look at a sad but true story that happened to a female friend of a machine friend of my Apple. It happened quite a while back, but my Apple sees the same thing repeating over and over again.

A new and major upgrade of a very heavy piece of software was announced, and review articles appeared in all the usual magazines. So, she promptly ordered one through her local dealer. They took a deposit, but six months later, no delivery. She then went to two large and distant cities, checking no less than seven different stores. Six of the stores told her flatly that the product did not exist. One of them tried to shove an inferior substitute product onto her that *Peelings*, in a fit of generosity, had rated a full "D" instead of the flat "F" it rightly deserved. The seventh store owner did admit that the product she needed had existed for quite a while, but he did not presently stock it.

So far, over half a year had gone by, much gasoline, time, and phone money down the drain, and still no product. So, with no reasonable alternative, she got on the piracy underground net, and, because she was now desperate for this code, she used the gold channel with (gulp) an "R3" priority. Two days later she had two copies, one by UPS blue label, and one by express mail, along with a complete list of all known bugs, use hassles, and application hints. All at a total cost of only five BEU barter exchange units. Had *any* dealer made *any* effort at all to close an immediate cash sale, this would never have happened in the first place.

• • •

Are you one of those still trying to get the miniassembler going on a II+ or IIe?

Forget it.

There's a brand new tool called the *Bugbyter* available on the latest versions of the DOS 3.3 toolkits. A full-blown and expanded version of the miniassembler is built into the Bugbyter, among its many other valuable features.

Turning to real assemblers, the latest toolkits (now part of Apple's *Workbench* series) also offer a total overhaul and major revision of good old EDASM. Which now includes all the goodies like macros, in place assembly, 80 columns, co-resident assembly, lowercase, execution time printouts, branch taken addresses, sweet 16 and 65C02 support, improved conditional assembly, plus much more. You simply use Apple Writer IIe helped along by WPL. This "new way" full screen editing is modeless and trivially easy.

By the most astounding coincidence, there is a book out called *Don Lancaster's Assembly Cookbook for the Apple II/IIe* and published by Howard W. Sams & Co., Inc., (Cat. No. 22331). This includes full details on "old" and "new" EDASM, along with secrets of "new way" editing. Needless to say, neither Sams nor I would be adverse to your latching onto one of these. If you can't find one of these around locally, try dialing (800) 428-SAMS.

• • •

Want full color for the MAC? How about *all* of the MacPaint features for the IIe? You can have both for only $3.42. Just plug a cable between the IIe game paddle port and the MAC's communication port. Add some simple machine language drivers that *swap* screen images on command, and you are home free.

• • •

Turning to this volume's enhancements, number nine is chock full of goodies on print quality. Included are ways to proportional space and microjustify your Apple Writer IIe, full *self-tutoring* glossaries for the Epson and Diablo, a sneaky way to upgrade to "camera ready" print quality, sources of low-cost hard-copy munchables, and some fully automatic WPL programs that let you take your *already existing* text files and upgrade them, invisibly and automatically.

The worst "feature" of the IIe is the intentional omission of an absolute monitor reset. In Enhancement 10, we find a simple "swap the chip," $6.00 mod that returns absolute and unconditional control back to you. There's no more "hole blasting" done when you try to reset. Your absolute control is there all the time, yet remains completely invisible until you ask for it. All diagnostics pass.

Easing up a little, Enhancement 11 is a set of playing aids for Castle Wolfenstein. You can simply and cheaply build these, and they are more than rugged enough for continuous use.

Our heaviest enhancement in this volume tears Apple Writer IIe from stem to stern. In Enhancement 12, you will find a complete, thorough, and detailed analysis of the single most-used and most-popular IIe program of all time. Included are lots of extra details on our super-powerful "tearing method," new tearing resources, and specific instructions for capturing your own IIe source code for custom mods.

One of the complaints of the last volume was that there was too much needed in the way of hardware mods. So, to round out this volume, Enhancement 13 shows you several ways of doing fast and exact field sync that takes zero hardware mods, besides being much faster, much simpler, and much more powerful than before. Included are some windowing software and a completely software-driven color killer.

You saw them here first.

As usual, all of the software presented here is available ready to run on a companion diskette. Low cost and unlocked, naturally. For those of you heavy into Apple Writer IIe, there are also separate AWIIe Toolkit packages available, a total of 16 diskette sides. The order blanks are in the back of the book.

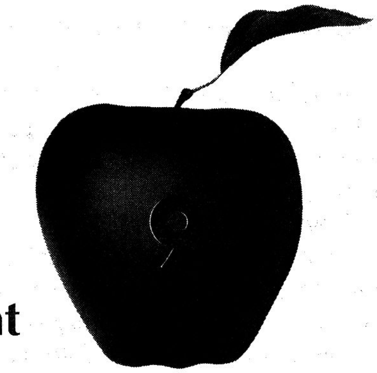
Also as usual, we have the Apple enhancers voice hotline going at (602) 428-4073. This is a combined service of the Gila Valley Apple Growers association and Synergetics. We might expand this to a bulletin board or possibly a CompuServe® service sometime soon. The double "focus" of the hotline currently involves machine language programming and Apple Writer IIe word processing.

Everything here is more or less my own work and is done without Apple's blessing or consent. Many useful comments on the AWIIe tearing were made by Bob Sander-Cedarlof. Moose choreography was by Elanior Fairley. Apple is a registered trademark of Apple Computer, Inc. So is Apple Writer. Castle Wolfenstein is copyrighted by the Castle Wolfenstein people. Anyone else I have run roughshod over, I usually mean their trademarked or copyrighted product, for which credit is hereby given.

DON LANCASTER

This book is dedicated to Carl and Jerry.
Yes, that Carl and Jerry.

This enhancement can help you improve print quality on most any word processor. Detailed examples use Apple Writer IIe.



Enhancement

MICROJUSTIFY AND PROPORTIONAL SPACE APPLE WRITER IIe

Print quality of most any word processor and printer combination can be dramatically improved by following a few simple rules. You can easily and automatically proportional space and microjustify Apple Writer IIe, improve underlining, do shadow printing, and lots more.

MICROJUSTIFY AND PROPORTIONAL SPACE APPLE WRITER IIe

Most hard copy from most personal computers looks just plain awful.

If I had my way, I would use a phototypesetter for all my computer rough drafts and "quick and dirty" internal printouts. Naturally, anything that went out the door should be done much better than this.

Gravure would be nice.

Which says that the first person to come up with a $200 phototypesetting or laser printing plug-in for an Apple will run away with a very large bag of marbles. Particularly if the font library is available for less than a dollar a whack.

It also says that those dot-matrix printer ads will sooner or later run out of new euphemisms for the word "atrocious" when describing their print quality. Also sooner or later, the daisywheel people will run out of weasel words that lie about their print speed. How about "paragraphs per century"? That ought to handle it once and for all.

Anyway, back to reality. You are probably stuck with a dot-matrix or a daisywheel printer at this point in time. What can you do to be sure you have the best possible print quality on everything you send out your door?

Let's look at some major ways you can improve the print quality of your particular word processor and printer combination. Then, as some specific examples, we'll find out how to add some real bells and whistles to an *Apple Writer IIe* and *Diablo 630* word processing setup. These bells and whistles will get you within shouting distance of "typeset" quality. With some care, all this can be handled fully automatically, even using your *already* existing word processor files!

Regardless of your word processor or printer, you'll find lots of ideas here you can adapt for your own use.

We will first look at four print quality rules that will apply to you regardless of what you are now printing with. After that, we'll find out extra-cost choices that will let you upgrade beyond your present print quality.

Here's our first rule . . .

Print Quality Rule Number ONE
Have the original image created by the original author match the final image seen by the final reader as EXACTLY as possible.

It's not just the words that count in the communication process. It is how those words are arranged and how they are viewed that is everything.

The closer your "rough draft" looks like the "final result," the better your print quality will be. And the better you will be able to balance the content and the appearance of what you are trying to create. And, most importantly, the fewer ways people between you and the final result can find to foul up the works.

Now, this is no big deal on a business letter. But, on anything like an article, a paper, or a book, this rule becomes crucial.

Make what you first type look as *exactly* as possible as what your final reader will see. Anything else in this day and age is not even absurd.

Here's our next rule . . .

Print Quality Rule Number TWO

Be sure you get ALL the parts needed for your printer. These are NEVER included in the purchase price.

Question: A *Diablo 630* printer lists for $1800 on dealer special. How much will this printer cost you? Answer: With lots of luck and some compromises, you might squeak by for less than $3000.

All of the essential and good parts needed for top print quality are purposely left out of most printer packages sold by most computer dealers.

First, you will need the *real* manuals for your printer. These include the service, maintenance, configuration, interface, and repair manuals. Not just that vapid whatever that got stuffed into the shipping box. Cost of these special manuals is typically $25 to $45 each. Normally, a dealer will go out of his way to *prevent* you from ever getting your hands on any of these, for it cuts dearly into his service work. You usually have to order directly from the factory.

The complete manuals are absolutely essential for (a) finding out the true capabilities of your machine, (b) maintaining print quality on a long-term basis, and (c) solving compatibility and use hassles.

Secondly, you will need some decent way of feeding paper. For daisywheels, this means a print tractor. A bidirectional tractor is needed for graphics and for top text quality. One-way tractors simply cannot hack it. No tractor at all is a cruel joke at best. Tractors are best bought used or surplus, since they are far cheaper this way.

Thirdly, you will need the special toolkit required to keep the printer alive. *Diablo* will gladly let you buy their little box with two funny screwdrivers, a tiny steel rod, some sticky glop, and two pieces of stamped metal in it. All this for a mere $75 plus shipping. Unfortunately, you must have these tools and parts to make any sane use of the printer at all.

Fourthly, you will need to find out about the extra-cost bells and whistles. Like an engine and wheels. On dot-matrix printers, this includes graphics ROMs, screen dumpers, and font downloaders.

On daisywheel printers, look for enhanced or expanded circuits that let you do microjustification, full word processing, vector graphics, PS table downloading, or otherwise add intelligence. As we'll find out later, the key trick is to let the word processor software do what it does best, let the printer firmware do what it does best, and then link the two as intelligently as possible.

Finally, you will probably want to build a silencer. Most any printer will drive you up the wall if you work anywhere near it. While the dot-matrix printers seem somewhat quieter than the daisywheels, their noise is higher pitched and much more stressful. If you think toolkits and manuals are priced out of sight, wait till you see the pricing on silencers. Hoo boy. It is best to build your own. Be sure to include a fan.

The bottom line is to find out what you *really* need to get your printer doing useful things, and then pick up as much of it as quickly and as cheaply as you possibly can.

On to our third rule . . .

Print Quality Rule Number THREE
Find sanely priced sources of word processing supplies.

All of the "munchables" that get gobbled up by your word processing system will eat you out of house and home if you let them, which either costs you plenty or else forces you to use second rate materials and supplies.

Tune yourself into reasonable and sane sources of paper, ribbons, diskettes, mailers, formfed checks, stationary, labels, and whatever. This will both save you money and give you a far better product out the door.

SEEDS and STEMS

Using the Monitor MOVE Routine

There is a powerful and useful move routine built into the Apple system monitor. More people seem to have more problems putting it to use than just about any other Apple feature. Here's how:

Put the destination low in \$42 and high in \$43.
Put the source start low in \$3C and high in \$3D.
Put the source end low in \$3E and high in \$3F.

♦ clear the Y register to #\$00! ♦

Then JSR \$FE2C to make your in-program move.

For instant hex-to-decimal conversions and back again, check into the hassle-free *Hexadecimal Chronicles* (Sams Cat. No. 21802).

You can move anywhere that does not overlap upwards without any problems. If upwards overlap is needed, do two moves, to and from an "outside" area.

You can clear or "replicate" any memory area by trying to move one byte up in memory. For instance, this routine quickly "empties" all of user RAM below DOS 3.3 to easily read "11" values:

```
* 0800: 11 <cr>
* 0801 <0800.95FEM <cr>
```

Here's a list of some of the supply sources I use . . .

Word Processing Supply Sources that I Use . . .	
New ribbons and paper:	QUILL CORPORATION 100 S. Schleter Road Lincolnshire, IL 60069 (312) 634-4800
Ribbon rewinding:	TORRES RIBBONS 416 East State Street Redlands, CA 92373 (714) 792-0831
Diablo parts:	THE PRINTER WORKS 1961 Alpine Way Hayward, CA 94545 (415) 887-6116
Printer bargains:	COMPUTER SHOPPER Box F Titusville, FL 32796 (305) 269-3211
Bulk 3M diskettes:	ALF PRODUCTS 1315F Nelson Street Denver, CO 80215 (303) 234-0871
Diskette Mailers:	CALUMET CARTON 16920 State Street S. Holland, IL 60643 (312) 333-6521
Checks and labels:	NEBS COMPUTER FORMS 12 South Street Townsend, MA 01469 (800) 225-9550

Once again, these are just the sources I happen to be using at this writing. Naturally, the instant I find some place that is better or cheaper, they will get my business.

Two tips. *Quill* has far and away the best pricing on ribbons and metal daisywheels anywhere, but be sure to wait for their monthly "loss leader" sales. What you need will usually come back around again in a few months. Secondly, *Calumet* calls their reusable diskette mailers a "#1 Stay-Flat Mailer." Pricing is—are you ready for this—under a dime each in quantity.

Always ask around for the best pricing and delivery on your word processing needs. Chances are you are presently being ripped off. As guidelines, medium-quality diskettes should cost you no more than $1.50 and Diablo film ribbons no more than

$2.75. Diskette mailers not over a dime. Metal daisywheels should not exceed $30.
Our fourth and final rule . . .

Print Quality Rule Number FOUR

Let the word processor program and the printer communicate with each other as intelligently as possible.

If you do not tell your word processor what is hung on its output as a printer, it will assume it is something worse than a Model 28 *Teletype®*, and you will get print quality that is both atrocious and slow. On the other hand, if you let your word processor and printer talk to each other at the highest possible level, you can optimize your results for superb quality.

The trick is to let the printer do what it does best, and let the word processor do what it does best. For instance, if you have a word processor with only medium-quality fill justification and a printer with full microjustification, have your printer and not the word processor do the justification for you.

There are two levels of communication involved . . .

Low-Level Communication —

Letting the word processor send stuff to the printer as quickly and as error-free as possible.

High-Level Communication —

Imbedding special commands into the text that activate special printer features when and as they are needed.

Low-level intelligent communication is nothing more than making sure your interface works. We won't worry too much about that here. Many, if not most, of the better quality printers communicate serially under standard RS-232.

Should you need extra RS-232 cables to extend a printer, you can build these yourself by buying press-on connectors and flat cables out of most any electronic hardware catalog. Cost is far less than ready-made cables, and the result is compact and flexible. Suitable cables may also show up as surplus bargains.

You should always try to communicate at the fastest possible baud rate, preferably 9600 bits per second. Otherwise time is wasted passing characters back and forth and your printer will have to wait every now and then for new characters. This becomes crucial on daisywheel graphic dumps.

Needless to say, both ends must use the same baud rate. Hidden switches can get flipped or software commands can get sent to adjust the baud rates.

You should also defeat any Apple video display echo unless you really need it. The reason is that the long screen scrolling times will further slow down any exchange of characters between your word processor and printer. The "old" IIe has an especially slow screen scroll.

And, most importantly, you should be sure you have handshaking between your printer and your word processor. If the printer ever gets behind, it must have some way

of telling the computer to stop sending characters for a while. The usual microcomputer way of handling this is with a "busy" signal line that goes from printer to computer. The busy signal holds up any characters being sent until they can be used.

The object of the game is to have the printer limit its own maximum speed, rather than slowing things down just to avoid handshaking.

Summing up . . .

> For a good serial interface —
>
> (1) Run at the fastest baud rate.
> (2) Defeat the Apple screen echo.
> (3) Provide "busy" handshaking.

As you have probably found out by now, "RS-232 Compatible" means only that you do not have to call the fire department before you plug two RS-232 interface connectors into each other. The compatibility in *no way* guarantees that the two devices will actually talk to each other or do what you expect of them.

The number and use of each wire in an RS-232 interface differs with each printer application. Fig. 9-1 summarizes the important RS-232 signal lines, while Fig. 9-2

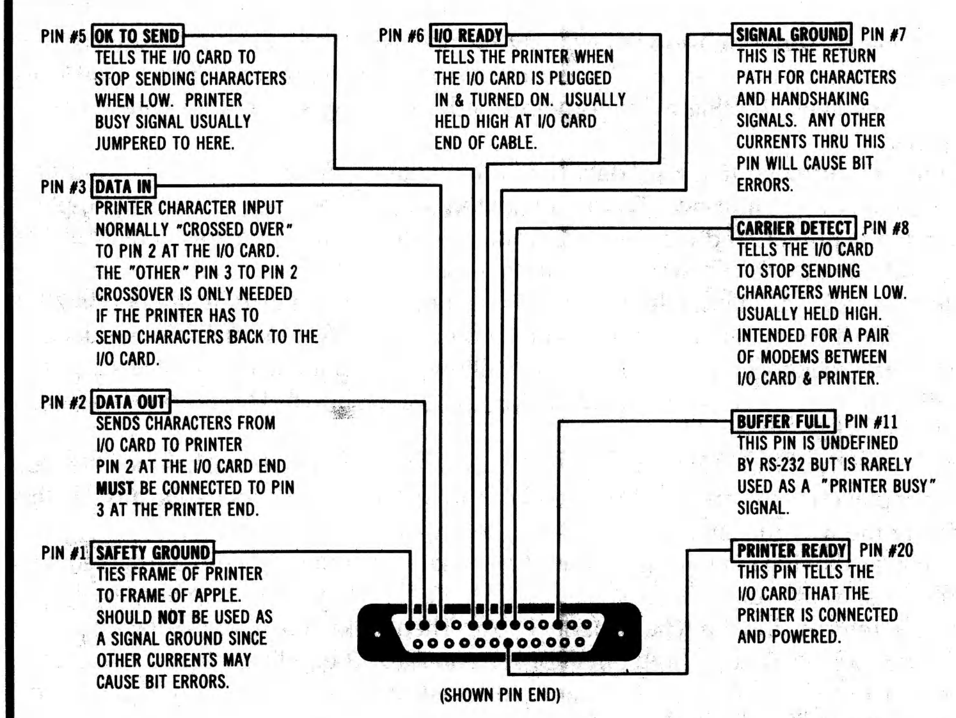


FIGURE 9-1. The RS-232-C pin connections normally used in a serial printer interface.

shows the interface I use between an Apple IIe with a *Mountain Hardware* AIO card in slot one. This interface lashup is more or less typical, but details may change with your needs.

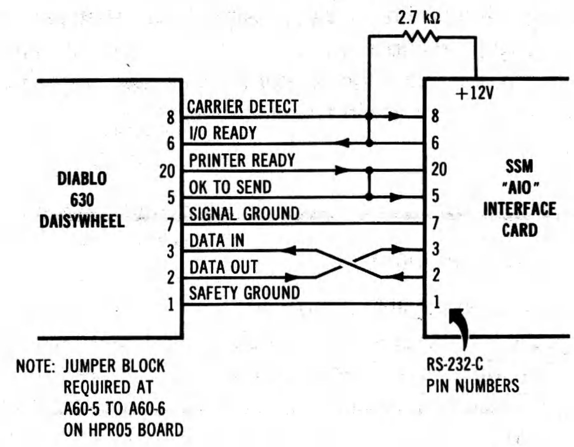


FIGURE 9-2. Examples of the "customizing" often needed to get an RS-232-C interface to work properly. This example links a Diablo 630 daisywheel to an SSM "AID" interface card.

In general, some "customizing" may be needed to get a serial RS-232 printer interface running the first time, if you use an oddball older printer or printer card. This usually involves crossing a pair of wires and jumpering some other wires or lines together.

Note in particular that the "data out" and "data in" lines, pins 2 and 3, must be crossed once and only once. The reason for this is that what is output from the Apple is input to the printer, and vice versa. Commercial "modem eliminators" are available that will cross these two wires for you. Pricing is unreal.

In many RS-232 interfaces, you can simply tie pins 6, 8, and 20 together. Pin 20 is usually the busy signal line from printer to computer. Unfortunately, this introduces a bug in the extended Diablo 630 that swallows two characters out of each buffer loading, so pins 6 and 8 are separately held active as shown. This is done by using a pullup resistor.

Incidentally, the circuit of Fig. 9-2 only works on a Diablo 630 that has the little blue jumper placed between pins A60-5 and A60-6 on the HPRO5 board. Details like this drive home why the special manuals are so important.

If you are having interface problems, first make sure your baud rates are the same on both ends, as are such things as the word length, the number of stop bits, the forced carriage returns, and the type of parity in use. Then make sure your interface has pins two and three crossed in both directions. As an RS-232 baseline, cross 3 to 2, 2 to 3, and separately jumper pins 6, 8, and 20 together. Next, try printing at the *lowest* possible baud rate, preferably 110 baud. This will separate any fundamental gotchas from handshaking problems. Finally, go up to full speed to resolve any handshaking problems. Handshaking problems usually won't show up until a few hundred characters have been properly printed.

If all that fails, use an oscilloscope or a voltmeter to check which lines are doing what to whom.

More information on interface fundamentals appear in the *TV Typewriter Cookbook, Micro Cookbook* Volume 1, and *Micro Cookbook* Volume 2, Sams Cat. Nos. 21313, 21828, and 21829, respectively).

That just about summarizes the four most important print quality rules and some interface guidelines. These ought to work for you, no matter where you are or what you have now in the way of hard copy.

FOR STILL MORE PRINT QUALITY

If you are willing to spend more to get more, here are some additional ways you can upgrade your print quality . . .

For Best Print Quality —

Use a daisywheel printer, rather than a dot-matrix printer.

Use a real daisywheel printer, instead of a toy one.

Use a metal daisywheel element, rather than a plastic one.

Use a proportional-spaced wheel, rather than a fixed-pitch one.

Use a printwheel font whose vibes match the image you are after.

Use a film ribbon, rather than a fabric one.

Use fresh ribbons from a quality source.

Use a bidirectional print tractor, rather than a unidirectional one.

Use the slowest possible printing speed with maximum settling time.

Use the highest quality paper that best suits the job to be done.

Use a ribbon and paper combination that work well together.

Use microjustification rather than full space justification.

Use your maintenance manual to keep the printer fine tuned.

Use a new platen, rather than one that is two years old.

Use every feature you can, so long as it improves your final product.

Some comments here. Daisywheel print quality is vastly better than dot-matrix print quality.

Period.

While even the toy daisywheels will do a reasonable printing job, only the "real" daisywheels can give you top-drawer quality. At this writing, there are only three

"real" and mainstream daisywheel sources. These are the *Qume* Sprint series, the *Diablo* 630, and the heavier *NEC* Spinwriters®. Technically, the *NEC* is really a thimble printer rather than a daisywheel, but you end up with essentially the same results and about the same print quality.

There is as much difference in print quality between a metal daisywheel element and a plastic one as there is between a plastic wheel and dot-matrix quality. Admittedly, the metal wheels have a much higher "first cost" and are very easily damaged; but for superb results, there is no contest.

Note that the print elements on metal daisywheels are thicker than those on plastic ones. Thus, the optimum printer hammer setting for metal is unsuitable for plastic, and vice versa. It is best to stick with all metal daisywheels, rather than continually readjusting your machine.

The best daisywheel elements will offer proportional spacing, rather than fixed spacing. This means that thin characters get printed close together and fat characters get printed far apart. With proportional spacing a capital "W" takes up more room than a lowercase "i". This is just like real printing, but unlike your usual typewriter. Proportional-spaced printing is far more readable. With proportional spacing, you can often cram more message in less space.

You will find hundreds of different daisywheels available. The type font you pick sets the overall "vibes" of your message and the tone with which it is to be received. Experiment to get the best overall results. I personally like the BOLD PS wheel for people-style communications, and the TITAN 10 for machine language dumps and other computer listings. Later on, we will find out how to automatically handle oddball spokes on offbeat wheels.

There is also no comparison between film and cloth ribbons. Film is sharper, blacker, and far better looking. High-quality fresh ribbons from a source you trust will give you better and more uniform results. Often, though, the "house brands" will be just as good and far cheaper than "genuine" name-brand stock. It pays to check. Never nurse a sick ribbon. Flush it as soon as it even threatens trouble.

Also uncomparable are two-way, or *bidirectional,* print tractors. The bidirectional tractor positions the paper far more precisely. These are essential for clean graphics or for printing pages with several text columns in them. It is a fairly simple matter for a bidirectional tractor to back up as much as a full page. Any reverse motion at all gets sticky fast with one-way tractors.

Just as some dot-matrix printers will give you higher quality by slowing down and printing more dots per character, some daisywheels will let you slow down your printing by increasing the carriage settling time. Increased settling time gives you more accurate character hits, which are essential for shadow printing and otherwise may improve results. Thus, it is often a good idea to run your final "out-the-door" copy as *slowly* as you possibly can.

Your choice of printer paper should be obvious. At the very least, use 20 pound, extra-white, microperf paper for anything except rough internal drafts. On sale, this stuff runs less than a penny a sheet. Where customer acceptance is critical, step up to classic laid papers or bond papers with a high rag content.

Should you be going to offset print for your final text, use a super-white, slick-surface, hard-coated stock for your camera-ready copy. The litho camera used to make printing plates will give you much sharper characters on this type of stock. Your local printer can put you on to sources. *Hewlett-Packard* plotter paper makes a fairly cheap but workable substitute.

Some of the newest daisywheels offer a "camera-ready" print mode. Usually, they just shift the ribbon into high gear so each character gets its own fresh chunk of ribbon for its own private use. Normally, there is only a one-fifth character advance of the

ribbon on each hit, so the price for "camera-ready" quality is higher ribbon use. A good tradeoff, but worthwhile only for your "final" copy.

You might be able to fake this "camera-ready" mode if your daisywheel offers downloading of custom proportional space/ribbon advance tables. Just change the table so each character uses enough ribbon for a guaranteed clean hit.

I've come up with a program called WPL.CAMERA READY that dramatically upgrades the final appearance of my Diablo, particularly on better quality paper. It works by giving the wheel extra time to settle down, and then whacks each character twice. All you have to do is run this program on your existing files just before printing. Ribbon cost is around double, and printing time is much slower, but the results are superb.

WPL.CAMERA READY is available as a bonus program on the companion diskette to this volume. See the order card in back of the book for full details.

Some ribbons work well with certain papers and poorly with others, so it is best to carefully test and then match your paper and ribbon to each other.

Microjustification means that you can fill out a line by uniformly expanding each space and, if needed, the space between letters. The adjustment can be as small as 1/120 of an inch, or a tiny fraction of the width of a full character. The fill justify option on most word processors only lets you do whole space justification. This creates an awkward "shading" across the text and is visually jarring. Microjustification also can let you individually adjust side-by-side characters so they look as natural together as possible.

This side-by-side adjustment is called *kerning*. More on this later.

Keeping your printer properly adjusted is very important. The print quality of just about any machine will deteriorate with time. Important things to do every few months on a daisywheel printer include "washing" the printwheels, adjusting the print hammer mechanism, freeing up the tractor, correcting the linefeed stepper backlash, and doing a general cleaning and lubrication. Once again, you'll find the maintenance manuals essential to help you do the job right. Special tools and gauges may also be needed.

It is also a good idea to replace the platen at least every two years. Do this whether it needs it or not. Naturally, if the platen looks bad at any time, replace it promptly.

Platens on personal computer printers tend to wear unevenly, since much of the copy consists of narrow listings or machine dumps. Sometimes a mildly worn platen can be flipped end for end to extend its life.

Finally, there is nothing worse than failing to use an existing printer feature that genuinely will help your hard copy. Be sure to learn each bell and whistle on *both* your printer and word processor.

Then figure out how to combine them in original and useful ways.

An Example

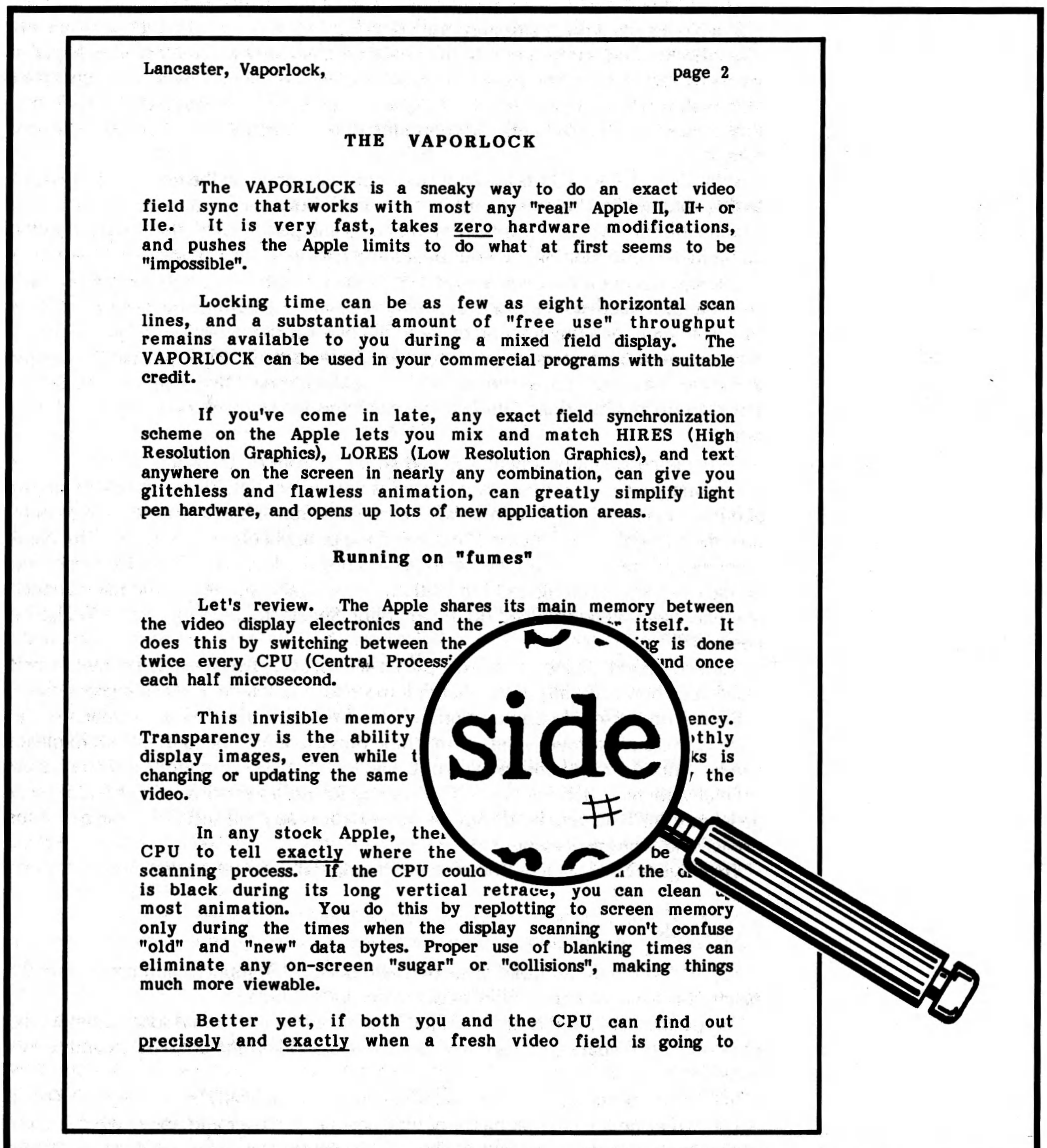
Fig. 9-3 shows an example of second-rate personal computer hard copy. Unfortunately, it is the best I know how to do at the present time.

As you can see, there is full microjustification and proportional spacing, improved titles, attractive spacing, hassle-free underlines, and kerning done between the "V" and the "A" in the title.

What you cannot see is that this was done fully automatically by starting with a stock *Apple Writer IIe* textfile that has very little special in it over and above what you are probably now using. With the process I'll be showing you here, you can easily and automatically print your *existing* and new textfiles to this quality level.

You can do all this without worrying about or even looking at the special imbedded instructions that handle the "magic" printing for you.

What you see here is an unretouched litho photo of the original daisywheel hard copy, except for the obvious games we played with the magnifying glass. The blowup inside the glass is also unretouched.



Lancaster, Vaporlock, **page 2**

THE VAPORLOCK

The VAPORLOCK is a sneaky way to do an exact video field sync that works with most any "real" Apple II, II+ or IIe. It is very fast, takes zero hardware modifications, and pushes the Apple limits to do what at first seems to be "impossible".

Locking time can be as few as eight horizontal scan lines, and a substantial amount of "free use" throughput remains available to you during a mixed field display. The VAPORLOCK can be used in your commercial programs with suitable credit.

If you've come in late, any exact field synchronization scheme on the Apple lets you mix and match HIRES (High Resolution Graphics), LORES (Low Resolution Graphics), and text anywhere on the screen in nearly any combination, can give you glitchless and flawless animation, can greatly simplify light pen hardware, and opens up lots of new application areas.

Running on "fumes"

Let's review. The Apple shares its main memory between the video display electronics and the ... itself. It does this by switching between the ... ng is done twice every CPU (Central Process... nd once each half microsecond.

This invisible memory ... ency. Transparency is the ability ... thly display images, even while t... ks is changing or updating the same ... the video.

In any stock Apple, the... CPU to tell exactly where the ... be scanning process. If the CPU could ... the d... is black during its long vertical retrace, you can clean ... most animation. You do this by replotting to screen memory only during the times when the display scanning won't confuse "old" and "new" data bytes. Proper use of blanking times can eliminate any on-screen "sugar" or "collisions", making things much more viewable.

Better yet, if both you and the CPU can find out precisely and exactly when a fresh video field is going to

FIGURE 9-3. Typical second-rate personal computer hard copy.

By the way, the wide margins and extra spaces between paragraphs in this example are a compromise that attempts to give the editor room to work, yet closely approximates one on one what the final reader will actually see on a book page.

Now, I won't for an instant claim that this is "good" print quality or that it approaches "typeset" quality. That would be almost as absurd as claiming that dot- matrix printing was legible or that daisywheel printing was fast.

What I do suggest, though, is that you use this example as a *minimum* quality standard. Don't let anything get out *your* door unless it looks *at least* this good. You can start with this as a "baseline" quality goal and improve things from there.

Let's find out how to get there from here.

IMBEDDING PRINT COMMANDS

There are several older ways of passing "high-level" commands back and forth between a word processing computer and a printer. Some of these older methods involve hardware switch flipping and some involve configuration jumpers. Others involve direct pokes or stores to certain memory locations.

Today, most printer intelligence is passed back and forth with easily used *imbedded printer commands . . .*

> Imbedded Printer Command —
>
> A "message inside a message" that tells the printer to start doing something special or different.

Let's look at three examples of imbedded commands.

The *Epson* command "[esc]4" tells the printer to begin printing in italics, while "[esc]5" turns italics off. Similar commands will pick the boldness of the printing, italicize or underline, or set the number of characters per inch vertically or horizontally.

We will use the *WPL method* of showing control keys here. Thus, "[esc]" means "press down the escape key," while "[L]" means "press down the control key, press and release the "L" key, and then release the control key."

Turning to daisywheels, the enhanced *Diablo* command "[esc]M" will turn on the microjustify feature, while "[esc]X" will turn microjustify back off again. Similar imbedded commands alter margins, spacing, graphics selections, proportional printing, shadow printing, and similar features.

Some plug-in interface cards will also need imbedded commands. More correctly, these are *communications* commands, rather than *printer* commands. One familiar example is the "[I]80N" command used by parallel interface cards to set the line width to 80 characters and cancel the Apple video echo.

If you are going to use all of the really neat features of your particular printer and interface, you have to be able to understand and use imbedded print commands. More importantly, you have to figure out how to let your word processor handle these commands for you.

Here's how to start . . .

> To understand imbedded commands —
>
> (1) Find a list of them.
> (2) Play with them one on one.
> (3) Add them to your word processor.

I like to do things by hand and by myself. This way, I force myself to think about what I am doing when I am exploring something new.

So, get a list of special commands for your printer, and then hand list them. Rearrange things in order of likely interest to you. After that, spend some time "exercising" each new feature. Do this until you thoroughly understand what each command does and what it can do for you. It is very important to check out each of these commands well ahead of time as a separate study, rather than trying to figure something out in the middle of some "real" printing.

Be sure to explore both what the imbedded command is intended to do as well as what it really can accomplish if you use it in new and unexplored ways.

How do you place your imbedded commands in your word processor text files?

That, of course, depends on your choice of word processor. With *Apple Writer* IIe, there are three different levels at which you can imbed print commands . . .

> **Imbedding Commands in Apple Writer IIe**
>
> Verbatim Method —
>
> Use the "[V]" command to place control characters into your text when and as they are needed.
>
> Glossary Method —
>
> Use the glossary to give you single-keystroke entries of long imbedded commands.
>
> WPL Method —
>
> Use the WPL word processing language to enter or strip whole documents of the needed commands.

The *Verbatim* method is the simplest. You use it to immediately put an occasional control character directly into your text. Use this method for practice and for seldom-used or oddball commands.

The glossary method lets you shorten each often-used series of imbedded commands into a single keystroke. This saves looking up and keying for commands that you use repeatedly. The glossary selections are put there in the first place by using the verbatim method. The glossary is then saved to disk for later reuse.

The WPL method is a super heavy. Under automatic program control, the WPL method can get a document spread over as many drives as you have in your system. It will then scan the document and automatically insert or remove any and all imbedded commands of your choice, any way you want.

One big plus of the WPL method is that you never have to look at imbedded commands or work directly with text that has lots of hard-to-read instructions imbedded in it. You take a plain, old text file, done the way you already know, and then use WPL to automatically imbed the commands *immediately before* each printing. Changes or corrections are always made to the original, pre-imbedded document.

The WPL method is particularly handy if you have to send your files to, say, both a daisywheel and a phototypesetter. Each of these will have its own set of wildly different imbedded commands. It is best to have your text files with a minimum of any imbedding at all, and then customize the files as needed for each output.

Let's look at each imbedding method in turn. If you are *not* using *Apple Writer* IIe, then try and find some "alike but different somehow" method that works for you. But, as you will quickly find out, there is nothing, repeat nothing, in the entire word processing world that can even hold a candle to WPL.

Verbatim Method

The [V] key tends to drive new users to *Apple Writer* IIe up the wall. If you accidentally hit this key combination, all sorts of weird things start happening to your text and you seem to lose control. Inverse "H" gets added every time you try to backspace, and every attempted deletion ends up *adding* a new character.

What's going on here?

Remember that we will use the [V] symbol to mean "hold down the control key, press and release the capital V key, and then release the control key."

When you do a [V], you tell your *Apple Writer* IIe word processor, "From now on until further notice, I want you to ignore all the control commands I give you except for [V] and [M]." "Instead, you are to *directly* imbed any other control keys into your text file."

Thus, on an accidental [V] hit, the left arrow gets ignored, but a backspace gets imbedded into your text as an [H]. Tabs become [I] and so on. The more you stir it, the worse it gets, since the *only* commands the program will recognize as commands are [V] and [M], the carriage return.

Unfortunately, there are several different ways of showing control commands. Table 9-1 gives you a list of the 33 ASCII control commands, their traditional names, and how they are keyed from your Apple. For instance, we see that ASCII code hex $11 or decimal 17 is called "DC1", short for "Device Control One", and is entered from your Apple keyboard by a [Q].

Note that any and all ASCII control commands can be entered directly from the IIe keyboard. Some, such as [Q] will need the control key held down. Others, such as [esc] or [tab] will *directly* generate the needed code, with or without using the control key.

While we are looking at tables, Table 9-2 shows us a table of "N" and "N – 1" values. Many intelligent printers expect to receive certain numeric values as an equivalent ASCII character. This is akin to the "CHR$" command in BASIC that sends out a carriage return "CR" for a "CHR$(13)", and so on.

The reason for this oddball turn of events is that a numeric value can be sent as a single ASCII byte, rather than needing two or three. There is also no guessing involved on whether a received "1" is really a "1", the start of a "12" or the very beginning of a "123" sequence. Thus, instead of a "101", you send a lowercase "f" instead. One fixed byte instead of three variable ones.

Sometimes, the actual or "N" value is used. Other times, an "N – 1" value is sent. Which one is used depends on the command and the printer. Line and tab values are often done with "N" commands, while VMI and HMI motions are often handled by

the "N−1" values. See the advanced manuals on your particular printer for full details.

We are purposely not going to study the individual commands of individual printers in depth. First, because you *must* do this on your own if you are to get the most bang for

Table 9-1. ASCII Control Codes Are Needed for Most Intelligent Printer Interfaces. They Can Appear in Many Different Forms. Here Is How to Key Them:

ASCII	Hex Code	Dec Code	IIe Keys	Original Use
NUL	$00	00	[@]*	Do nothing or null
SOH	$01	01	[A]	Start of heading
STX	$02	02	[B]	Start of text
ETX	$03	03	[C]	End of text
EOT	$04	04	[D]	End of transmission
ENQ	$05	05	[E]	Enquiry
ACK	$06	06	[F]	Acknowledge
BEL	$07	07	[G]	Bell or alarm
BS	$08	08	[H]	Backspace
HT	$09	09	[I]	Horizontal tab
LF	$0A	10	[J]	Line feed
VT	$0B	11	[K]	Vertical tab
FF	$0C	12	[L]	Form feed
CR	$0D	13	[M]	Carriage return
SO	$0E	14	[N]	Shift out
SI	$0F	15	[O]	Shift in
DLE	$10	16	[P]	Data link escape
DC1	$11	17	[Q]	Device control #1
DC2	$12	18	[R]	Device control #2
DC3	$13	19	[S]	Device control #3
DC4	$14	20	[T]	Device control #4
NAK	$15	21	[U]	Negative acknowledge
SYN	$16	22	[V]	Synchronous idle
ETB	$17	23	[W]	End block transmit
CAN	$18	24	[X]	Cancel
EM	$19	25	[Y]	End of medium
SUB	$1A	26	[Z]	Substitute
ESC	$1B	27	[{]	Escape
FS	$1C	28	[|]	Form separator
GS	$1D	29	[}]	Group separator
RS	$1E	30	[^]	Range separator
US	$1F	31	[_]	User separator
DEL	$7F	127	DELETE*	Delete

Equivalent keys:	LEFT ARROW	=	[H]	=	BS
	TAB	=	[I]	=	HT
	DOWN ARROW	=	[J]	=	LF
	UP ARROW	=	[K]	=	VT
	RETURN	=	[M]	=	CR
	RIGHT ARROW	=	[U]	=	NAK
	ESCAPE	=	[{]	=	ESC

Gotchas: Many Apple uses set the ASCII most significant bit. To set the MSB, add hex $80 or decimal 128 to above values.

* - NUL ($00 or $80) and DEL ($7F or $FF) are reserved for internal use by Apple Writer IIe.

Table 9-2. Many Intelligent Printers Need Command Values Passed to Them by Using the Numeric "N" or "N – 1" Value of an ASCII Character. Here Is a Complete Table:

"N – 1"	"N"	SEND	"N – 1"	"N"	SEND	"N – 1"	"N"	SEND
0	1	[A]	44	45	–	88	89	Y
1	2	[B]	45	46	.	89	90	Z
2	3	[C]	46	47	/	90	91	[
3	4	[D]	47	48	0	91	92	/
4	5	[E]	48	49	1	92	93	]
5	6	[F]	49	50	2	93	94	^
6	7	[G]	50	51	3	94	95	_
7	8	[H]	51	52	4	95	96	`
8	9	[I]	52	53	5	96	97	a
9	10	[J]	53	54	6	97	98	b
10	11	[K]	54	55	7	98	99	c
11	12	[L]	55	56	8	99	100	d
12	13	[M]	56	57	9	100	101	e
13	14	[N]	57	58	:	101	102	f
14	15	[O]	58	59	;	102	103	g
15	16	[P]	59	60	<	103	104	h
16	17	[Q]	60	61	=	104	105	i
17	18	[R]	61	62	>	105	106	j
18	19	[S]	62	63	?	106	107	k
19	20	[T]	63	64	@	107	108	l
20	21	[U]	64	65	A	108	109	m
21	22	[V]	65	66	B	109	110	n
22	23	[W]	66	67	C	110	111	o
23	24	[X]	67	68	D	111	112	p
24	25	[Y]	68	69	E	112	113	q
25	26	[Z]	69	70	F	113	114	r
26	27	[{]	70	71	G	114	115	s
27	28	[|]	71	72	H	115	116	t
28	29	[}]	72	73	I	116	117	u
29	30	[^]	73	74	J	117	118	v
30	31	[_]	74	75	K	118	119	w
31	32	(space)	75	76	L	119	120	x
32	33	!	76	77	M	120	121	y
33	34	"	77	78	N	121	122	z
34	35	#	78	79	O	122	123	{
35	36	$	79	80	P	123	124	|
36	37	%	80	81	Q	124	125	}
37	38	&	81	82	R	125	126	~
38	39	'	82	83	S			
39	40	(	83	84	T			
40	41	)	84	85	U			
41	42	*	85	86	V			
42	43	+	86	87	W			
43	44	,	87	88	X			

"N – 1" values are often used by HMI and VMI motion commands.
"N" values are often used for lines/page and tab values.

ASCII values of $00, $7F, $80, and $FF are reserved for internal use by Apple Writer IIe and should not be imbedded into text files.

the buck out of your printer. Second, because there are so many printers and countless variations out there. And, last of all, because the final user need not worry about such things if he uses the invisible and automatic methods we are about to develop here.

Back to the program. How do you use [V]?

Say you want to switch your *Epson* to print italics. This means you want to imbed an "[esc]4" into your text. To do this using [V], first go to the place in the text where you want to imbed the command. Then type "[V] [esc] [V]4". The first [V] says to *verbatim* enter the escape key into the text. The second [V] says to quit imbedding funny commands and switch back to normal word processor use of the control commands.

Note in this example that we need a plain, old "4" and not a "[4]". Watch this detail very carefully. Provide control characters *only* when they are called for.

As an enhanced *Diablo* example, the command "[esc]M" turns on the micro-justification. To imbed this in your document, go to the right place, then type "[V] [esc] [V]M," and you are home free.

Obviously . . .

> Do NOT forget to use the second [V] when you finish imbedding a command!

Let's look at another simple example of the verbatim method.

One of the weaknesses of *Apple Writer* IIe's underline mode is that there is no direct way to underline up to a period, comma, or question mark. At least not without also underlining the punctuation or adding at least one unwanted space.

There is both a "micrometer" and a "sledgehammer" solution to this problem.

The "micrometer" way is to imbed a backspace *after* the trailing underline token but *before* the comma or period. For instance, you type "zorch\ [V] [H] [V]." to underline the "zorch" but not the period. The backspace swallows the space forced by the underline processor.

Unfortunately, this is only a "perfect" fix in the left justify mode. If you are using AWIIe's fill justify mode, you may have to play some very fancy right margin games as well.

The imbedded backspace does work, however, on any printer that can recognize a backspace. The imbedded backspace is a simple fix to the one problem that seems to bother beginning *Apple Writer* IIe users the most.

The "sledgehammer" solution is to imbed commands that turn the printer's underlining off and on. This lets the printer do its own much more flexible underlining. The usual result is a cleaner underline because the printer will step up the ribbon advance. Imbedded backspaces are also not needed this way.

Better yet, do you really want to underline? How about a double strike, a font change, a switch to italics, or a shadow print instead?

Solutions . . .

> **Three ways to let Apple Writer IIe underline up to a comma or period —**
>
> (1) Imbed a backspace before the punctuation.
>
> (2) Use the underline feature of the printer instead.
>
> (3) Substitute shadow printing, bold, or italic fonts.

There is a subtle gotcha involved with [V].

If you really want to imbed a [V] into a textfile, you have to get sneaky. The "one-line" entry mode that lets you create a glossary will let you directly generate a [V], but the "full editing" normal operating mode will not. One simple answer is to put a [V] manually into your glossary, and then call it over and over again as needed into your text. The [F], or find-and-replace, command also works.

There are two other imbeddings that may give you fits.

If you imbed a reverse slash and if you are using the same symbol for your internal underlining, you may end up with serious problems. The Diablo command to disable backwards printing needs the reverse slash. It pays to go out of your way to never use reverse slashes for anything but an underline command. Among other reasons, this symbol is not available on all daisywheels.

The magic sequence "(<" also must be avoided. This will try to turn on the footnote machine.

So how did I just print it?

Imbedding With the Glossary

The intermediate way to imbed print commands with *Apple Writer IIe* uses the glossary. This way, a single and meaningful key hit imbeds the whole command for you. No muss, no fuss, no bother. For instance, you can use "<open-apple>I" to start italics on your *Epson,* and "<open-apple>i" to turn them back off.

Big "I" turns the italics *on*. Little "i" turns the italics back *off.*

Or, on an enhanced *Diablo*, an "<open-apple>J" starts microjustifying and "<open-apple>j" stops it.

Big "J" to turn the microjustify *on*. Little "j" to shut the microjustify *off.*

By now, you are probably up to your ears in *Epson MX-80* glossaries. Program 9-1, called EGLOSS, gives you yet one more.

This one has made all the keystrokes as meaningful and as easy to learn as possible. A *built-in* single-key help screen is included that leaves your text unaltered. There's also lots of room to add your own custom code.

Few people realize that glossary entries can be used to directly execute WPL and other machine commands in Apple Writer IIe. This is in addition to the usual use of entering text into your main text file, which is the secret to the self-prompting features of EGLOSS, a feature that leaves *all* of the others out in the cold.

In AWIIe, any control characters imbedded in a glossary will do their usual editing function. Any control characters imbedded in a glossary that are preceded by a [V] and followed by a [V] will actually get imbedded in your text. A carriage return is automatically substituted for each "]" in the glossary.

Directly entering a tutorial help screen out of a glossary seems to take around 20 seconds and messes with your textfile. Both are unacceptable. What this glossary does instead is it reloads a portion of itself *only to the screen*. This displays the help screen that is tacked onto the end of the glossary. Total time to display is about 3 seconds. Erasure is instant, and your main textfile is untouched.

Typing "<open-apple>z" gives you the tutorial help screen. Typing [return] exits you to your *undisturbed* workfile.

Alternately, typing "<open-apple>Z" both gives you a help screen *and* saves a copy of the glossary to the currently active disk drive. Use this as a hassle-free way to transfer the glossary to each disk as needed.

The save-before-loading "<open-apple>Z" glossary option works even if you have changed diskettes and drives. It also puts a free copy of the glossary on each and every diskette that is likely to need it.

PROGRAM 9-1 Epson MX80 Formatting Glossary With Tutorial

```
?       Applewriter IIe/Epson MX80 Formatting Glossary
?
A[V][esc]#[V]
a[V][esc]>[V]
B[V][esc]G[V]
b[V][esc]H[V]
C[V][esc][O][V]
c[V][esc][S][V]
D[V][esc][N][V]
d[V][esc][T][V]
E[V][esc]E[V]
e[V][esc]F[V]
F[V][esc]C[V]
f[V][esc]C[@][V]
G[V][esc]L[V]
g[V][esc]K[V]
H[V][esc]J[V]
h[V][esc]2[V]
I[V][esc]4[V]
i[V][esc]5[V]
J[V][esc]N[V]
j[V][esc]O[V]
K[V][V]
k[V][V]
L[V][esc]=[esc][J][V]
l[V][esc]>[esc][J][V]
M[V][V]
m[V][V]
N[V][G][V]
n[V][G][V]
O[V][esc]U[@][V]
o[V][esc]U1[V]
P[V][esc]9[V]
p[V][esc]8[V]
Q[V][V]
q[V][V]
R[V][esc]=[V]
r[V][esc]>[V]
S[V][esc]S[@][V]
s[V][esc]S1[V]
T[V]D[V]
t[V]D[V]
U[V][esc]-1[V]
u[V][esc]-[@][V]
V[V][esc]1[V]
v[V][esc]0[V]
W[V][esc]Q[V]
w[V][esc]Q[V]
X[V][esc]@[V]
```

PROGRAM 9-1 cont

```
x[V][esc]T[V]
Y[V][V]
y[V][V]
^[V][L][V]
<[V][H][V]


         Epson MX-80 Open-Apple Formatting Commands:
........................................
.......................................

(A) ascii b8 as is   (H) height custom *
(O) two way print    (V) VERY tight 7/72
(a) ascii b8 one     (h) height normal
(o) one way print    (v) very tight 1/8
(B) bold print on    (I) italics on
(P) paperout sense   (W) width column *
(b) bold print off   (i) italics off
(p) paperout ignore  (w) width column *
(C) compressed on    (J) jump perf on *
(Q)   (spare)        (X) off all modes
(c) compressed off   (j) jump perf off
(q)   (spare)        (x) off sub/super
(D) doublewide on    (K)   (spare)
(R) reset B8 = 1     (Y) yourstuff on
(d) doublewide off   (k)   (spare)
(r) reset B8 = 0     (y) yourstuff off
(E) emphasized on    (L) linefeed w/rst
(S) superscript on   (Z) tutorial + save
(e) emphasized off   (l) linefeed only
(s) subscript on     (z) tutorial only
(F) fmlngth lines *  (M)   (spare)
(T) tab set *        (^) formfeed
(f) fmlngth inchs *  (m)   (spare)
(t) tab set *        (<) backspace
(G) graphics 960 *   (N) noisy bell
(U) underline on
(g) graphics 480 *   (n) noisy bell
(u) underline off

Capital letter is "on", "yes", "above", or "more".
For full features, the AWIIe NULL patch is needed.

* - follow with ASCII value(s).  (See Epson us
                         er manual for details.)
```

PROGRAM 9-1 cont

```
Z[P]nd[F]<>>><><A][Q]FEGLOSS][L]EGLOSS<>     <ls.)>><\][P]yd]
z[L]EGLOSS<>     <ls.)>><\]
```

Gotchas: Pairs of brackets mean control commands. [esc] = escape key, [Q] means "<ctrl>Q", etc. Note that any isolated brackets really are isolated brackets.

The line preceding the tutorial screen must have four spaces on it. "Z" and "z" selections must NOT precede the tutorial or they will find themselves.

Eighty-column tutorial text lines have been split. Entries (A), (H), (O), and (V) all go on one line. The dot row is also one continuous line, as is the line starting with "* - Follow".

By the way, certain features of EGLOSS will not be available to you unless you make a simple mod to AWIIe that lets you imbed NULL commands in a document.

Which leads us to a large can of worms . . .

*NULL*IFYING AWIIe AND SUPERSCRIPTING EPSONS

There are two "features" of Apple Writer IIe that quickly lead to some printer compatibility hassles . . .

Two Bad "Features" of stock AWIIe

(1) ASCII characters of $00, $7F, $80, and $FF are not allowed in an AWIIe text file.

(2) Any imbedded printer commands that AWIIe does not recognize get treated as real characters and will shorten lines that are fill justified.

We can call the first one the "Epson NULL" problem, since it restricts sending ASCII $00 or NULL commands needed for Epson features on older printers, particularly underline and superscript. The second one is obviously the "short line" problem.

Note that the ASCII $80 character was easily entered in old Apple Writer 2.0, simply by doing a [@]. This character is specifically *excluded* from AWIIe.

Automatic "repair" programs, called the AWIIE NULLIFIER and the AWIIE STRETCHIFIER, are included as bonus programs on the support disk for this volume. Free copies are also available directly from the helpline. In addition, one version of the NULLIFIER patch appears in Chart 9-1.

As near as I can tell, this NULL patch is benign. You lose an oddball feature of the DELETE key, which you should not be using in the first place. Remember that you can *undo* a deletion that was done by "<open-apple>-<left arrow>", while the DELETE key is forever.

The auto-repeat will nail you every time if you try to use the DELETE key.

Even without this NULL patch . . .

NEVER use the AWIIe DELETE key, for anything, anyplace, ever!

Use <open-apple> and <left arrow> instead, since any possible damage can later be undone.

Chart 9-1 NULLifying Apple Writer IIe

While the ASCII $00, or NULL command is easily placed in an older Apple Writer 2.0 file by doing a [V] [@] [V], NULLs are specifically excluded from Apple Writer IIe.

Among their many other uses, NULLs are needed for some HIRES graphics dumps and for Epson underlining or superscript features.

Here is a two-byte patch that lets you imbed NULLs into your AWIIe files.

WARNING: The following descriptions work ONLY on EXACTLY the DOS 3.3e "E" and "F" versions of Apple Writer IIe, circa March of 1983. Use the "tearing method" to adapt to any updates or revisions.

USE ONLY YOUR THIRD BACKUP COPY FOR THESE PATCHES!

For the "E" version used with a normal 80-column card:

(1) Boot normal DOS 3.3
(2) BLOAD OBJ.APWRT] [E,A$2300
(3) PRINT PEEK (9956) (quit if not 15)
(4) POKE 9956, 00
(5) PRINT PEEK (18621) (quit if not 214)
(6) POKE 18621, 00
(7) UNLOCK OBJ.APWRT] [E
(8) BSAVE OBJ.APWRT] [E,A$2300,L$2F5A
(9) LOCK OBJ.APWRT] [E

For the "F" version used with an extended 80-column card:

(1) Boot normal DOS 3.3
(2) BLOAD OBJ.APWRT] [F,A$2300
(3) PRINT PEEK (10116) (quit if not 21)
(4) POKE 10116, 00
(5) PRINT PEEK (18998) (quit if not 214)
(6) POKE 18998, 00
(7) UNLOCK OBJ.APWRT] [F
(8) BSAVE OBJ.APWRT] [F,A$2300,L$30D3
(9) LOCK OBJ.APWRT] [F

Test either version by booting the modified AWIIe disk. Start typing random stuff into a new text file. Imbed a NULL by doing a [V] [@] [V] or a [V] [2] [V]. The NULL should appear as an inverse "@".

A bonus program called AWIIE NULLIFIER appears on the companion diskette to this volume. It will automatically patch both versions for you, after it verifies that the code is safe to alter.

You can imbed a NULL using EGLOSS, or else by using [V] [@] [V], after making the patch.

To avoid disallowed $FF values during a HIRES screen dump, use six rather than seven pins on the dot-matrix printhead. More on this in a future enhancement.

A DIABLO 630 GLOSSARY

Program 9-2 is an enhanced *Diablo 630* glossary that provides nearly all of the available features plus lots of room for macros of your own. Once again, an instant access help screen is provided, keying on "<open-apple>z" for help only, or on "<open-apple>Z" for a combined save and help screen.

Why the WP enhanced *Diablo* 630?

First, because it is what I use. If I find something that is faster, cheaper, quieter, more reliable, and more flexible than the 630, and if it also has better long-term print quality, I'll quickly buy it.

Only it hasn't happened yet and it doesn't seem likely for a while.

Secondly, because the other two daisywheel heavies, *Qume* and *NEC*, are "alike but different somehow" in their use of imbedded commands. And finally, because many of the new toy daisywheels claim to be more or less "Diablo compatible," whatever that means.

One glossary tip that a lot of people miss . . .

> In Apple Writer IIe, the <open-apple> key does the same thing as [G].

This little trick saves you lots of keystrokes. I usually put the entire top of the fancy thought boxes into the glossary under a space key. One pressing of "<open-apple>(space)" instantly gives you the entire top of a text display box.

One sloppy keystroke replaces around 65 careful ones!

There are two very minor gotchas. First, don't use " / " as a glossary key since "<open-apple> / " gets you the main help screen instead. If you like, comments can be inserted into a glossary with your choice of "?", "*", or " / " as the first character in the comment string.

Secondly, note that the "z" and "Z" help screen commands must *follow* the help screen text itself. The reason for this is that there are search strings in both of these commands. If the command precedes the screen text, the command will find *itself* instead of the tutorial text.

Nearly anything you can do with the [V] method can be done quicker and simpler with the Glossary method. Just use the [V] method to create the glossary *once*. Then use the glossary as often as needed. Once your glossary is created, nontechnical people will find it very simple and easy to use.

As usual, all of the programs here are available and ready to go on the companion diskettes. See the response cards and ordering information in the back of the book.

PROGRAM 9-2 Diablo 630 Formatting Glossary With Tutorial

```
?       Applewriter IIe/Diablo 630 Formatting Glossary
?
A[V][esc][K][V]
a[V][esc][U][V]
B[V][esc]O[V]
b[V][esc]&[V]
C[V][esc]=[V]
c[V][esc]X[V]
D[V][esc]$[V]
d[V][esc]X[V]
E[V][esc]0[V]
e[V][esc]9[V]
F[V][esc]A[V]
f[V][esc]B[V]
G[V][esc]3[V]
g[V][esc]4[V]
H[V][K][V]
h[V][tab][V]
I[V][esc]%[V]
i[V][esc]N[V]
J[V][esc]M[V]
j[V][esc]X[V]
K[V][esc][Q][V]
k[V][esc]X[V]
L[V][esc][L][V]
l[V][esc][H][V]
M[V][esc]^[V]
m[V][esc]_[V]
N[V][esc]T[V]
n[V][esc]S[V]
O[V][esc]D[V]
o[V][esc]U[V]
P[V][esc]P[V]
p[V][esc]Q[V]
Q[V][esc]7[V]
q[V][esc]X[V]
R[V][esc]/[V]
r[V][esc]\[V]
S[V][esc]W[V]
s[V][esc]&[V]
T[V][esc]-[V]
t[V][esc]1[V]
U[V][esc]E[V]
u[V][esc]R[V]
V[V][esc][J][V]
v[V][J][V]
W[V][esc]Z[V]
w[V][esc]Y[V]
X[V][esc]2[V]
```

PROGRAM 9-2 cont

```
x[V][esc]S[V]
Y[V][V]
y[V][V]
^[V][L][V]
<[V][H][V]
,[V] [esc]Q[esc]l[H][tab] [V]
.[V][esc]P[V]

         Diablo 630 Open-Apple Formatting Commands:
........................................
........................................

(A) absolute VTAB*  (H) vertical tab
(O) offset hl up    (V) vertical up
(a) absolute HTAB*  (h) horizontal tab
(o) offset hl down  (v) vertical down
(B) bold print on   (I) improve quality
(P) proportional on (W) wheel spoke $7F
(b) bold print off  (i) normal quality
(p) prop. space off (w) wheel spoke $20
(C) centering on    (J) justify on
(Q) quit printing   (X) clear tabs
(c) centering off   (j) justify off
(q) quit wp modes   (x) clear HMI
(D) dash hyphen on  (K) kerning set *
(R) reverse <- on   (Y) yourstuff on
(d) dash hyphen off (k) kill v margin
(r) reverse <- off  (y) yourstuff off
(E) west margin set (L) lines/page set*
(S) shadow on       (Z) tutorial + copy
(e) east margin set (l) little backspace
(s) shadow off      (z) tutorial only
(F) funny ribbon on (M) motion VMI **
(T) tab vert. set   (^) formfeed
(f) black ribbon on (m) motion HMI **
(t) tab horiz. set  (<) backspace
(G) graphics on     (N) north margin
(U) underline on    (,) dots start
(g) graphics off    (n) south margin
(u) underline off   (.) dots end

Capital letter is "on", "more", "above", "vertical", or "right".

*  - follow with ASCII value    (use Table 9.2 " N ")
** - follow with ASCII value-1  (use Table 9.2 "N-1")
```

PROGRAM 9-2 cont

```
Z[P]nd[F]<>>><><A][Q]FDGLOSS][L]DGLOSS<>     <-1")>><\][P]yd]
z[L]DGLOSS<>     <-1")>><\]
```

Gotchas: Pairs of brackets mean control commands. [esc] = escape key, [Q] means "<ctrl>Q", etc. Note that any isolated brackets really are isolated brackets.

The line preceding the tutorial screen must have four spaces on it. "Z" and "z" selections must NOT precede the tutorial or they will find themselves.

Eighty-column tutorial text lines have been split. Entries (A), (H), (O), and (V) all go on one line. The dot row is also one continuous line.

Another Glossary Example

Sometimes, you may want to combine several commands at once into a "macro" for your glossary. Let's look at a macro example of *Diablo* glossary use.

One of the stickiest problems you will come up with if you try proportional spacing is that columns of figures will get messed up, as will alignments of addresses, or anything else that is supposed to be "lined up" in the middle of a long string of characters.

The usual way out of this bind is to set tabs that align any needed columns. Most proportional-spaced fonts have constant values for all the numbers, so numbers will align on proportional spacing if they start at the same position.

Unfortunately, the hex characters "A" through "F" do not. The solution is to print columnar hex values at constant spacing, set so wide that the letters do not run into each other. Which is a royal pain, but well worth it for top appearances.

Note that the tabs must be set *inside the printer*, and not inside the word processor, for only the printer knows where it is on a proportionally spaced line at any instant.

Fig. 9-4 shows us a worst-case example of exiting proportional spacing at a totally random point on a line, putting down an aligned row of dots at fixed pitch, and then resuming proportional space with an aligned column.

If you try this without getting sneaky, you will find the dots "microstaggered" all over the lot, for each exit of a proportional-spaced line can end up anywhere with respect to fixed pitch.Thus, on exiting to 12 pitch, you can end up in any of 10 possible dot positions, only one of which ends up properly aligned.

The solution to this? Two glossary entries, called "," and "." You do "," first, then your dots, then the "." glossary entry. Or, if you just want spaces, you still do the same thing, putting "," before the string of spaces and "." after. Do this while holding down the <open-apple> key.

What happens is this: After the book name, you cancel proportional spacing. Then you set a tab. The tab sits at a constant value in 12-pitch, rather than the microjustified position you ended up in at the end of the proportionally spaced character string. We don't have to worry about exactly where this tab sits.

This newly set tab takes the end "slop" out of the line. Next, you back up one character and then tab to your newly set tab location. All of which eliminates any microstaggering and gets you back in line with plain old 12-pitch spacing.

Next, you put down your dots at 12 pitch, nonproportional. After that, you switch back to full proportional for the rest of the line.

The DON LANCASTER Library

Apple II & IIe Assembly Cookbook	# 22331
Active Filter Cookbook	# 21168
CMOS Cookbook	# 21398
Cheap Video Cookbook....................	# 21524
Enhancing your Apple II vol I	# 21822
Hexadecimal Chronicles	# 21802
Micro Cookbook I (Fundamentals)	# 21828
Micro Cookbook II (Machine Language)	# 21829
Son of Cheap Video	# 21723
TTL Cookbook	# 21035
TV Typewriter Cookbook	# 21313

FIGURE 9-4. One of the trickiest things involving proportional spacing is proper column alignment.

One minor gotcha. When you do this, some of the columns may *still* not be aligned. This is caused by lots of wide characters in one name and a few thin ones in another. But, the column alignment will now be off by *whole* 12-pitch character spaces one way or the other. Simply do a printout and paint green all the lines that are short and paint pink all the lines that are long, using page highliters. Then add or remove a dot or two or a space or two as needed. It all comes out even.

Reviewing, the first part of the line gets put down in full proportional spacing. Then a tab is set to take out any microstaggering. Then you move to that tab. Then you put down dots, and then you go back to full proportional spacing. To do this, you put down your first character string, a comma glossary entry, a string of periods, a period glossary entry, and your final listing. Then you dump to a printer, and add or remove any whole dots as needed.

That quick and that easy.

The glossary command of "," combines moving forward one space, switching off proportional spacing, setting a tab, backspacing, and then going to that same tab you just set, and then doing another space.

Not bad for one keystroke.

If you like, you can follow this with spaces instead of dots. And that the "<open-apple>." glossary entry does is get you back into proportional spacing. The "<open-apple>P" command would do the same thing.

Chances are that there are other things you might like to do with one keystroke instead. There is lots of room left in the glossary for anything you like along these macro lines.

Oh, yes. Almost forgot.

There are, of course, several obscure bugs in the *Diablo 630* firmware, just as each and every decent piece of firmware anywhere unavoidably and inevitably has obscure bugs in it. As near as I can tell, these bugs come from sloppy initialization or else from two routines using the same variable for two different uses.

Two examples. On the first right-to-left pass after setting up a full microjustify, underline commands on the 630 either get ignored or else are badly garbled. So do certain spoke commands involving "[esc]Y" or "[esc]Z". Unfortunately, you have to restart microjustifying on each and every paragraph, since the microjustifying is far too aggressive to use on the last line of a typical paragraph. And, getting back to our

example, the first time you feed those assorted "," glossary commands to your 630, it will misuse them.

The sledgehammer way around the centering hassle is to print each paragraph that has *any* underlining or funny spokes in it left to right only. The way around the dot adjusting routine is to "adjust" a line of several blank spaces before you try using real characters on real lines.

What I am leading up to is . . .

> Strange bugs are likely to crop up if you try using printer firmware in new, obscure, or oddball ways.
>
> Expect this, and be willing to experiment your way to a solution.

By the way, I have an older but WP enhanced 630. The newer ECS and API machines may have these bugs corrected. On the other hand, there are almost certainly new bugs introduced to replace the fixed ones. In general, the number of bugs in a system will always *increase* with time.

As you go through the *Diablo* examples, you are likely to find some really weird and wondrous things that I have done. Things like a positive linefeed followed immediately by a negative one. Stunts like this *seemed* to eliminate a bug that *seemed* to exist at the time. The logical basis is that something got initialized this way that didn't before.

Other times, it is sort of like whacking a mule twice with a 2 by 4. The first time gets his attention; the second accomplishes what you really have in mind.

If it works, use it.

Iffen ain't broke, don't fix it.

Now for the really fun part. The part that lets *Apple Writer IIe* run away with all of the marbles. Of course, I'm talking about the neat things that start happening when you begin automatically . . .

IMBEDDING WITH WPL

Program 9-3 gets us started on the heaviest method of upgrading print quality.

This program is called WPL.FORMAT DIABLO 630. It takes a text file and imbeds magic values in just the right places to radically upgrade your print quality. Text is printed with proportional spacing, rather than the uniform spacing you would seem to be stuck with. This means that a lowercase "i" is more closely spaced than an upper case "W", and so on.

The results almost look typeset.

Secondly, all full lines that were fill justified are now fully microjustified, automatically expanding and compressing text so it will nicely and smoothly fit. Where before you could only add whole spaces, you can now microjustify in spaces of 1⁄120 of an inch. Since the Diablo microjustify is far too "aggressive" on incomplete text lines, the last line in a paragraph is automatically left justified instead. To not do this would spread out the last line unacceptably. Unless you are working with very narrow columns.

As additional features, any center justified text is automatically shadow printed, stretched out slightly horizontally, and tightened vertically. Any underlined text is

PROGRAM 9-3 Diablo 630 "Full Features" WPL Formatter

```
 pnd
 ppr[L]
 pprFormatting for Diablo 630 Special Features:
 ppr.........................
 ppr
 ppradjusting squashticity
 b
 psx4 squashticity factor
 f<>.dbl<>.dbl>.rm+(x)<
 y?
 p
 e
 f<<>.rm-(x)><
 y?
 p
 pprimproving underline
 b
 f<<.ut><
 y?
d f<\< [esc]E<
 y?
 pgod1
 pgod2
d1 f<\<[esc]R <
 y?
 pgod
d2 b
 pprfix (.,)
 f<[esc]R .<[esc]R.<a
 p
 p
 f<[esc]R ,<[esc]R,<a
 p
 p
 f<.[esc]R<[esc]R.<a
 p
 p
 f<,[esc]R<[esc]R,<a
 p
 p
 f<. ><.><a
 p
 p
 pprfixing underline bug
 e
e u
 f<[esc]E<
 p
 pgoe1
 pgoe2
```

PROGRAM 9-3 cont

```
el f<><>[esc]\<
 y?
 f<[esc]E<
 p
 pgoe
e2 p
 pprfix bold PS wheel
 b
 f<!<^<a
 p
 p
 b
 f<]<[esc]Z<a
 p
 p
 b
 f/</\/a
 p watch ut!
 p
 b
 f/[/</a
 p
 p
 b
 f/|/!/a
 p
 p
 b
 f/>/`/a
 p
 p
 b
 f/@/[esc]Y/a
 p
 p
 pprsetting title microjustify
 pas==$A right margin = 61
 b
 f<>>.db2><>.db2>[esc]X [esc][tab]z [esc][tab]$A [e
            sc]0 [esc][tab]z [esc][tab][A] >[esc]M<
 y?
 p
 f<>>.ff<>[esc]X>.ff<
 y?
 p
 ppradjusting byline
 b
 f<  by D<[esc]Xby D<
 y?
 p
```

PROGRAM 9-3 cont

```
 f<><>[esc]M<
 y
 P
 pprshadowing main title
 psx5 lines in title
 psx+1
 b
 f<>.db2<
 P
 pgoa3
 pgoa2
a3 u
 f<><
 p
 u
 p
a f<><>[esc]%[esc]W<
 y?
 pgoa1
 pgoa2
a1 psx-1
 pgoa
a2 b
 pprtightening spacing
 b
 f<>.db1<
 p
 pgob
 pgob3
b h
 h
 f<>>><>[esc]U><
 y?
 pgob1
 pgob3
b1 f<>>><>>[esc]D><
 y?
 pgob2
 pgob3
b2 f<>>><
 p
 pgob
b3 p
 pprsetting body microjustify
 pasF=$A .rm70
 b
 f<>.db1<
 P
 f<>.fj><[esc]X [esc]N [esc][tab]z [esc][tab]$A [esc]0 [e
             sc][tab]z [esc][tab][A][esc]/>.1j >[esc]M<A
```

PROGRAM 9-3 cont

```
 P
 pprjustifying insets
 pas<=$A .rm60
 b
 f!#.lm+12![esc]X [esc][tab]z [esc][tab]$A [esc]0 [e
             sc][tab]z [esc][tab][A] [esc]M#.lm+12!A
 P
 pprfixing insets
 b
 f<>* <>[esc]X* [esc]U[esc]D[esc]M<A
 P
 pprshadowing titles
 b
 f<>.dbl<
 P
 pgoc
 pgocl
c h
 h
 f<>.cj><>.cj>[esc]\[esc]X[esc]W[esc][Q][B][esc]%  <
 y?
 p
 f<><[esc]N[esc]P><
 y?
 h
 f<>.cj<
 p
 pgoc
cl p
 pprfixing paragraph ends
 b
 f<>.dbl<
 P
 f<.><[esc]X.[esc]7[esc]/>[esc]M<a
 p
 b
 f<>.dbl<
 p
 f<^><[esc]X^[esc]7[esc]/>[esc]M<a
 p !=^ bold ps
 b
 f#?%#[esc]X?[esc]7[esc]/%[esc]M#a
 p
 b
 f<`><[esc]X`[esc]7[esc]/>[esc]M<a
 p >=` bold ps
 pprstretch blurb
 psx6 lines
 b
 f<>.db3><>.db3>[esc][J][esc][J]<
```

PROGRAM 9-3 cont

```
  y?
  p
  pgof
  pgofl
f f<><>[esc]U <
  y?
  p
  psx-l
  pgof
fl p
  pin[G][G][G]--- detail work filename? ---> =$d
  psz+l  for wpl supervisor
  pas$d =$d
  pcs/ /$d/
  pgog
  pdo$d
g pqt
```

Gotchas: Pairs of brackets mean control commands. [esc] = escape key, [L] means "<ctrl>L", etc. Note that any isolated brackets really are isolated brackets.

Heavily indented lines are a continuation of the previous line and must be entered without intervening spaces or carriage returns.

Fixed right margin values of 70 for body text, 60 for insets and 61 for a special title box are presently built into this program. These values can be patched where shown and as needed, or prompting can be added.

This WPL program assumes a Diablo 630 printer with an enhanced HPRO5 board having full word processing features. The program MUST be customized if any other printer is to be used. Certain features may not be available on other daisywheel configurations.

Be sure to follow the use rules in the text!

underlined with Diablo's underliner. Many of the apparent AWIIe bugs either disappear or else become moot with this program.

Lots of other features are available in the various support modules. Best of all, you can take what you want and leave what you don't want, customizing things to suit your own particular word processing needs.

Most importantly, the *Diablo* formatting takes place *immediately before printing*. There is no need to imbed or read all those weird and mysterious commands in your working text files. What you do is take your plain, old text file, do a "[P]DOWPL.FORMAT DIABLO 630", wait a minute or two for the tweedle, and then print.

Please note you have to have the expanded WP options HPRO5 board in your *Diablo* 630 for this program to work fully. Note also that the commands may have to be changed to suit other daisywheels.

There are usually two steps to formatting a document. The general, or WPL.FORMAT DIABLO 630 part does everything it can to format any old document. Then a new WPL.DETAIL WHATEVER takes over and handles anything oddball or specific that your current document may need in the way of special treatment.

We will shortly see detailed working examples of many different WPL support modules. Before we begin, please note that you are looking at what I am using for me. If you don't like what you see, then rearrange things to suit yourself. The beauty of WPL is its extreme flexibility. In fact, I know of no other computer language available anywhere ever that has such powerful editing features.

Every attempt has been made to keep anything special out of your unformatted text files. But, there are some simple rules you will have to *exactly* follow if you want these ready-to-go modules to work for you. Naturally, you are free to either obey the rules or else change the modules.

These are the present text file restrictions . . .

Text File Rules for Automatic Formatting

1. The machine must be less than 85% full since your text file will get longer.
2. No footnotes are allowed, unless you split the WPL programs into smaller chunks.
3. An imbedded ".db1" must precede the body of what you want printed with fully justified paragraphs and improved titles.
4. Imbedded ".db2", ".db3", etc., lines are needed to point out any areas that need special or detailed treatment.
5. Each and every center justified line must be immediately preceded by a ".cj", even with multiline centered titles.
6. Present text body margin settings are .lm7 and .rm70. Program 9-1 is very sensitive to right margin settings and must be customized on any change.
7. The reverse slash must be used only for underline on/off calls. No backspaces are to be imbedded at the end of an underline call.
8. Any paragraph that follows a centered title or an inset box must be preceded by a blank line followed by a ".fj".
9. All imbedded commands must always be done in lowercase.
10. All right margin changes must be relative rather than absolute.

As you can see, there is very little that is different from what you are probably now doing. Just be sure to put a ".db1" at the start of the body of your work, have a ".cj"

immediately before each and every title, and have a blank line and a ".fj" preceding every entry into plain, old paragraphs from anything else. Use only lowercase imbedded commands.

Each WPL module starts with a comment line that explains what that module does and ends with the last statement before the comment line that starts the next module. In general, you can add and remove modules any way you like, although the order of some modules is critical.

Here's a rundown of WPL.FORMAT DIABLO 630 . . .

WPL

WPL is a supervisory language that runs an executive controlling program. It does almost everything a person can do, given a long and detailed enough list of which keys to press in what order. The way a WPL program is activated is by a [P], the DO command, and the program name.

You write WPL programs exactly the same way you write any text file. You test and debug them the same way you would test and debug any computer program. Full details on what WPL is and how it works appears in the *WPL Programming Manual*. Please note that WPL is intimately linked to *Apple Writer* IIe.

Owning AWIIe and not using WPL is like buying a Porsche just to listen to its radio. You automatically exclude yourself from well over 95% of the potential of AWIIe if you do not become thoroughly WPL literate.

To understate, WPL is "not easily" moved to another word processor. I know of no other word processor available at any price that has anything remotely as flexible or powerful as WPL available for it.

Squashticity

During printing, the *Diablo* will be in its full microjustify mode, while *Apple Writer* IIe will run left justified. When in left justify, AWIIe tends to shorten lines by a few characters, since, more often than not, the word wraparound will move the last word down to the next line.

If left alone, this means that the average line will appear "stretched out" during full microjustify. Imbedded commands, particularly underlines, will also tend to stretch lines. Stretching means that the line will be the same length as the others, except that the individual characters will be further apart.

The optional "squashticity" factor tells AWIIe to give you longer lines than the *Diablo* would normally use. I like a factor of plus four or so, which is how much the AWIIe right margin is advanced during formatting. This means that the *average* line will end up neither squashed nor expanded. Note that the *Diablo* fill justify can expand or contract lines as needed. It is "elastic" in both directions, just like an already-stretched rubber band.

The optimum squashticity will give you the best overall appearance. It also will ease any problems AWIIe has in shortening lines during command imbedding, although the AWIIE STRETCHIFIER patch does a much better job of this.

Squashticity is adjusted by finding the ".db1" body marker and following it with a .rm + 4 or plus whatever. One gotcha. All right margin commands that follow must be relative.

Improving Underlining

There are several undesirable features of the AWIIe underliner, so it is best to transfer this task to the 630 firmware. You get faster operation, blacker and more

uniform underlines, and no hassles underlining up to punctuation. You can also underline and proportional space without ratty results.

To upgrade, the reverse slash underline token is cancelled and replaced with an empty ".ut". Then, pairs of reverse slashes are found. The first is replaced with a space and a "start underline" command. The second of each pair is replaced with a space and a "stop underline" command. This continues for all underline commands in the text.

Next, the "stop underline" commands are checked for either periods or commas. If the period or the comma is outside the underline, the trailing space is dropped. If the period or the comma is inside the underline, it is moved outside. Either way, you end up with your underline up to a period or a comma but stopping before the punctuation. This module also makes sure there is no space following the period at a paragraph end.

As we mentioned, there is a bizarre underline bug in the 630 that louses up underlining on the first "backwards" carriage run after a change to microjustify. A sledgehammer solution is used in this module. The document is scanned backwards. For every paragraph with *any* underlining in it at all, the entire paragraph is printed left to right only.

Several gotchas. You must use this underline repair module *before* you inadvertently insert any reverse slashes into the text by later modules. Reverse slashes might be needed by margin settings, cancellations of right-to-left carriage motions, or redefined spokes on oddball daisywheels. Right now, you cannot underline up to a question mark or exclamation point, although you could easily add this feature if you think it is important.

Using italics instead of underline is available on the 630-ECS. You also could do italics by wheel swapping, but this takes a lot of time and dedication. Other viable alternatives to underlining are to use doublestrike or shadow printing instead.

Redefining Wheel Spokes

As you no doubt have already found out, not all daisywheel elements have all their characters coded the same as does the *Apple* IIe keyboard. Some spokes may be missing, while others will print the wrong character. Yet other spokes will be available that have no key for them. The usual result is some wildly wrong punctuation and symbol errors.

The TITAN 10 wheel almost matches the Apple keyboards and AWIIe. All of the keys print the expected symbols in the expected way. You can also print a cents symbol by using "<open-apple>w" and a closing single quote by using "<open-apple>W".

Incidentally, these "two hidden characters" are typical of most 96 spoke wheels. There are two symbols on a 96 character wheel that cannot be directly printed. The reason for this is that there are only 96 printable ASCII characters, and the $20 space and $7F delete are reserved. To print the hidden characters, you use imbedded commands that I have coded into the Diablo glossary as "w" for the $20 wheel spoke and "W" for the $7F wheel spoke. These also will get fouled up on the first backwards 630 pass after a switch to microjustification.

Anyway, if you try to use the BOLD PS printwheel, you will find 15 problem spokes. These spokes will either print the wrong character or will print the right character on the wrong code. Seven of these are coded wrong, while the remaining eight will print a wildly different symbol.

Wrongly coded BOLD PS spokes include the opening and closing brackets, the opening and closing carats, the exclamation point, the vertical line, and the "@" symbol. Missing entirely are cents, reverse slash, up carat, tilde, squiggly brackets,

opening single quote, and closing single quote. The only serious loss here is the up carat, which limits your use of this wheel on some program listings.

But, proportional spaced listings are bound to lead to serious problems anyway. Use the fixed pitch TITAN 10 wheel instead, whenever you are doing program listings and dumps.

The eight "new" BOLD PS spokes are nothing to get excited about. These are listed in the usual printwheel catalogs. The degree symbol can sometimes help round the edges of big characters and other graphics dumps. Filled in, it makes a nice bullet.

What this module does is redefine the seven wrongly coded spokes. The document is scanned, once for each of the seven spokes. Each spoke is replaced with its correct coding. After this is done, the formatted listing will *look* wrong, but it will print properly with a BOLD PS wheel.

Gotchas here include that this module only works correctly for the BOLD PS printwheel. It also introduces some reverse slashes, so all the underline fixing must be done before this module is used. There is also no attempt to use any of the "new" spokes. If you really must have a copyright, trademark, double underline, degree symbol, or whatever, you can pick them up separately with your detail work, remapping them any way you care to.

Another major gotcha. Once the spokes are redefined, the new characters will affect all of the modules that *follow* this one. Thus, when you get around to searching for end-of-paragraph exclamation points, or "please don't justify" right carats, you have to search for the already remapped equivalent character.

Main Title Rework

I use a main title that is inside a box formed from rows of dots. At present, the title must be preceded by a blank line followed by a ".db2" marker. The present right margin is 61.

This module sets up the words inside the title box for full microjustification and proportional spacing. Then the byline that follows the main title is unjustified, keying on "by D". Finally, five lines in the main title are shadow printed. For best shadow printing, the carriage settling time is also increased.

This is admittedly a rather specialized module. You might like to rework it to suit your own private "title page" needs.

Tightening Spacing

Titles in the body of your text will often look better if they are spaced 1½ lines apart, instead of 2 full lines. This module finds each and every sequence of 3 successive carriage returns in your body, and converts them into 2½ carriage returns. The result is that all your titles get spaced by 1½ lines vertically.

What gets sticky fast is that AWIIe software is keeping track of page lengths. It knows nothing about half line feeds. This means you have got to keep the position on the page the same as where AWIIe thinks you ought to be. Otherwise, you'll get a bad case of "page creep," where your headers and footers will keep wandering up or down with respect to the page breaks.

The solution sounds hairy, but it works. On the first tightening, three carriage returns are replaced with three carriage returns and a negative half line feed. On the second tightening, three carriage returns are replaced with two carriage returns and a positive half line feed. The result, for each pair of tightenings, is one return removed from the text file and one line removed from the paper. It all comes out even, with no page creep.

One minor gotcha. You might end up with a rare shift of half a line on your page. This happens if an odd number of tightenings takes place on a page. The next page straightens things back out, so there is no page creep. Careful use of conditional form feeds will eliminate this. Chances are nobody would notice anyway.

Body Microjustify

The body of your text is intended to start at position 7 and end on position 70. This module finds all ".fj" commands in the body. It then converts them to ".lj" for AWIIe and substitutes full 630 firmware proportional microjustify.

The module sets the 630 margins and switches on 630 justify while switching off AWIIe fill justify. AWIIe ends up thinking it is left justifying text. The Diablo then takes the single-line text string and does a full microjustify on it.

There are several gotchas. Each paragraph in the body that is preceded by something funny, such as a title or an inset, must have a blank line and a ".fj" immediately before it. Paragraphs that follow paragraphs need no special treatment.

The right margin setting *must* be changed if you move your right body margin. To simplify this, the WPL string $A is used. You then get the character you need out of Table 9-2 and substitute it. We will later see a user prompting way of handling the same task.

Insets

As you might have guessed by now, I like to use a lot of insets or "thought boxes." The theory here is that your key points can be made to a reader who is quickly scanning your message. You can also integrate your text and visuals more closely this way. These thought boxes always have a border made of all asterisks. I try to make most of them 40 columns wide, a leftover from good, old AW1.0 days. These are all preceded by a blank line and a ".lm+12".

What gets sticky here is that you want to microjustify and proportional space the *contents* of the thought boxes while maintaining a smooth left margin to the printing *inside* of the box. Fortunately, the 630 firmware is reasonably good at justifying part of a line.

Fig. 9-5 shows us the problem. You want to justify from the *second* printing character from the left and not from the left border. Otherwise, you end up with a ragged column inside the box.

The module first sets the 630 right margin to 60 and then sets up a full microjustify for the box. Then, each and every line in the box has its fill justify cancelled until after the first asterisk. Two whacks with a 2 by 4 are needed to get the 630's attention, so cancelling line feeds in both directions is used as a sloppy bugstomper. There is normally no paper motion when this happens. The result is that the justification does not begin until the first printing character *after* the box border.

Lots of gotchas here. Each inset must be preceded by a blank line and a ".lm+12". Each must be exactly 40 spaces wide, and each must have a right margin at 60. Any special columnar material or hex addresses will need special detail work. Asterisks must presently be used for the box border.

Naturally, you are free to change any of this anyway you like. What I am showing you is what I use for me. The beauty of WPL lies in its extreme flexibility. Rearrange it to suit yourself.

Any way you like.

Just don't expect my code to automatically do exactly what you ask of it if you aren't going to play by my rules.

Imbedding Commands in Applewriter IIe

•••••••••••••••••••••••••••••••••••••

Verbatim Method -

Use the "<ctrl>-V" command to plant control characters into your text when and as they are needed.

Glossary Method -

Use the glossary to give you single keystroke entries of long imbedded commands.

WPL Method -

Use the WPL word processing language to enter or strip whole documents of the needed commands.

FIGURE 9-5. Handling insets and "thought boxes" can get rather tricky when you proportional space and micro-justify at the same time.

Shadowing Titles

Titles are shadowed by using shadow printing. Shadow printing whacks the character, moves 1/120 of an inch and then whacks it again. For optimum results, you also have to slow the machine down while you are doing this. You also will want to spread the title out somewhat.

This module goes through the body and finds each ".cj". It then cancels any remaining modes, sets up the slower shadow printing, while slightly spacing out the characters for that line. Normal speed is resumed at the end of the line.

The [B] following the "[esc]Q" gives you 2/120 of an inch extra spacing between characters. This makes up for the space lost in double whapping, and gives you some additional character spread. For less spread, use [A]. For more, use [C], [D], etc.

Gotchas here include the need to precede *each* line to be centered by a ".cj". The ".cj" must be the last thing before the title, and separate ".cj" commands must precede *all* lines of a multiline title.

I elected to use the AWIIe centering, rather than the 630 centering, since it seems much easier to use and control. One disadvantage of this is that some titles may not center themselves exactly. This is caused by the proportional spacing and becomes bad on very long titles with lots of capital letters in them. Some adjustment is built into this module, but you may have to touch up your centering here and there with your detail work.

The other option of using the Diablo centering will give you exact centering, but leads to all sorts of margin setting hassles. So far, I have used the AWIIe left margin and paragraph margin. The 630 left margin is left on column zero.

Actually, the BOLD PS wheel tends to overdo long sequences of all capital letters anyway. You tend to use fewer "all caps" title strings with this wheel.

Fixing Paragraph Ends

The 630 microjustification is extremely aggressive. It will both crunch and stretch by an amazing amount. Such aggressive justification is needed for very narrow columns. But, on book-width lines of 63 characters, most last lines of most paragraphs will get stretched out ridiculously far and will look awful.

To beat this, this module cancels the microjustification on the last line of each paragraph. What happens is that the body is scanned for a period followed by a carriage return. This is unique to a paragraph ending. That line then has its microjustification cancelled. With the right amount of squashticity and careful use of hyphens, all of your paragraphs lines should look fairly uniform.

Note the 2 by 4 technique here. Whack it a good one for openers. Then clobber it again. The only nice thing you can say about this coding is that it works.

Just watch out for splinters.

Since some sentences end in question marks or exclamation points, these are also scanned and corrected. So is a right carat, which lets you fake centering of an editorial comment without enhancing it as you would a regular centered title. Note that the right carat and the exclamation point have been redefined here, assuming a BOLD PS wheel. Other wheels will need special treatment.

There is one very annoying gotcha. It is fairly easy to get some spaces following your last paragraph period but before the carriage return. This is easy to let happen on rework and changes. I tend to do this every time I break a long paragraph. Single spaces before carriage returns are caught back in the underline module, but multiple ones are not. If you have a last paragraph line that refuses to unjustify, use a glossary command to search for hidden spaces.

In fact, it is best to *always* scan your copy before you do any formatting. Otherwise, you will get final paragraph lines stretched margin to margin, even if they only have a few words in them.

Stretching a Blurb

I call a *blurb* those lines that follow the main title and explain what the text is all about. These look good when they are microjustified to the same width as the main title and spaced out vertically by 1½ spaces each.

This module finds a ".db3" start-of-blurb marker and then adds an extra half-line feed to each of the six blurb lines. After that, two negative full line feeds and a form feed are used to reset the page position.

Gotcha: If you have fewer than six lines in your blurb, provide blank lines after your blurb but before the form feed to add to a total of six. For more than six lines, change the (x) line counter and add one negative line feed for each *two* blurb lines.

Detail Work

That should just about complete the automatic formatting that will handle most of what you need in the way of automatic formatting and justification.

Note that these formatting modules do not pay much attention to the *content* of your text files, so long as you obey some fairly simple rules. You are thus free to take older and existing text files, and then make a few very minor and low-key changes to them. Then, you can invisibly and automatically upgrade your print quality, just by running a WPL program or two immediately before printing.

If you are a print quality perfectionist, there will *always* be some loose ends left over that apply to the one particular document you are printing. So, when the formatter is finished, it asks you for the name of a detail work file. It then does the detail work needed to put the final touches on one specific document. Note that custom detail work will only apply to one specific text file.

Program 9-4 shows us two examples of detail work.

One of these is called WPL.DETAIL VAPORLOCK. A second is called WPL.DETAIL E9.

I'll leave these modules for you to puzzle over.

In WPL.DETAIL VL, one module improves the title centering. The next one unjustifies a reference bibliography at the end of the text. After that, one of the boxes that has some columnar material in it gets repaired so that the columns are aligned. Finally, the "V" and the "A" in the title are kerned so that the top of the "V" overlaps the bottom of the "A".

In WPL.DETAIL E9, the first module fixes up the little "configuration" box that starts off this enhancement. Some titles are stretched out, and then a wider thought box is improved. The suppliers list is also cleaned up some. A perfect cleanup was purposely not done, since the copy need not be camera-ready.

It is normally a good idea to *avoid* doing any detail work at all, unless superb final quality or "camera-ready" copy is really needed. The best approach is to "take half and leave half." Repair anything obviously bad, but don't spend hours and hours removing a glitch that nobody cares about anyhow . . .

> Avoid doing any detail work that is not really needed.
>
> You can waste monumental amounts of time on things nobody will notice.

As before, iffen it ain't broke, don't fix it.

As an aside, it gets tricky to dump a glossary or a WPL program to a printer, because the imbedded control commands will take over and start doing some really nasty things. Ugly even. A bonus program called WPL.LISTER solves this hassle for you and appears on the companion diskette to this volume. It was used to help create all the program listings for this module.

Note that WPL.LISTER does not find imbedded backspaces, frontspaces, or start-of-footnote commands. These must still be hand patched when and where they crop up. The program works by scanning for control characters, and then replacing them with the WPL notation, such as an [esc] for the escape key, or a [Q] for a DC1.

Ironically, WPL.LISTER is not very good at listing itself, although it does a beautiful job on practically everything else.

FINAL TOUCHES

Because of all the bells and whistles, you may find WPL.FORMAT DIABLO 630 taking several minutes to format a long text file. You will probably also find it wasting

PROGRAM 9-4 Two Examples of "Detail Work" Custom Formatting

```
 p
 p  630 detail formatting file  WPL.VAPORLOCK
 p       (requires DGLOSS on same disk)
 p
 ppr  improve title center
 b
 f<THE  VAPORLOCK    <THE  VAPORLOCK<
 y?
 p
 f<TH<  TH<
 y?
 ppr  unjustify references
 p
 b
 f/.rm+40/
 p
 f/[esc]M//A
 p
 ppr  fix columnar list
 b
 f/.db5/
 p
 pgog
 pgog2
g psx3
gl f<[esc]U[esc]D[esc]M  <<
 y?
 f<$<[esc][tab]& [esc]U[esc]D[esc]M$<
 y?
 psx-1
 pgog1
g2 p
 ppr  kerning VA
 e
 qedgloss
h f<VA<
 p
 pgoh1
 pgoh2
h1 u
 gl
 gl
 f<><>[esc]\<
 y?
 pgoh
h2 b
 ppr[G][G][G]
 pqt
```

• • • • • • • • • • • • • • • • • •

PROGRAM 9-4 cont

```
 p
 p  630 detail formatting file  WPL.DETAIL E9
 p
 ppr  justifying use box
 pas)=$A .rm36
 b
 f!#.db4![esc]X [esc][tab]z [esc][tab]$A [esc]0 [e
             sc][tab]z [esc][tab][A] [esc]M#.db4!A
 p
 p
 ppr  fixing use box
 psx7
 b
 f/.db4/
 p
 p
a f<>??<>??[esc]X[esc]U[esc]D[esc]M<
 y?
 p
 p
 psx-1
 pgoa
 p
 ppr  spreading titles
 b
 f/     MICROJ/MICROJ/
 y?
 p
 f/IONAL     /IONAL/
 y?
 p
 f/   SPA/SPA/
 y?
 p
 f/IIe   /IIe/
 y?
 p
 psx4
b f/  Print Q/Print Q/
 y?
 p
 f<  *><*>
 y?
 p
 psx-1
 pgob
 ppr  justifying wide insets
 p
 b
```

PROGRAM 9-4 cont

```
 pas@=$A .rm65
 b
 f!#.lm+6![esc]X [esc][tab]z [esc][tab]$A [esc]0 [e
             sc][tab]z [esc][tab][A] [esc]M#.lm+6!A
 p
 ppr  fixing supply list
 b
 f/.db8/
 p
 pgozl
 pgoz
zl psx7
y f<:=*><
 p
 p
 f<     *>< *><
 y?
 psx-1
 pgoy
z b
 ppr[G][G][G]
 pqt
```

Gotchas: Pairs of brackets mean control commands. [esc] = escape key, [G] means "<ctrl>G", etc. Note that any isolated brackets really are isolated brackets.

Detail work should be avoided.

These WPL programs assume a Diablo 630 printer with an enhanced HPRO5 board having full word processing features. The program MUST be customized if any other printer is to be used. Certain features may not be available on other daisywheel configurations.

Be sure to follow the use rules in the text!

time on things you neither want nor need. To speed things up, Program 9-5 is a shortened formatter called WPL.FORMAT D630 NOFRILLS.

This one "only" does the squashticity, underline improvement, spoke fixing, space tightening, body microjustify, title shadowing, and paragraph end fixing. As an added convenience, you are prompted for your body right margin setting, letting you change margins without changing the code. The no-frills version may be all you ever need, and it formats much faster with fewer complications.

Almost certainly, the first time you try to use any of these routines, they will either do nothing at all or else will thoroughly plow up your document. Expect this and carefully work with each module until you understand what it can and cannot do. Be willing to experiment ahead of time, rather than fighting a deadline on something critical. Don't forget that ".db1" before the body of your text.

PROGRAM 9-5 Diablo 630 "No Frills" WPL Formatter

```
 pnd
 ppr[L]
 pprDiablo 630 "no frills"  Formatter:
 ppr..................................
 ppr
 pprPlease enter right margin setting as one SINGLE CHARACTER.
 ppr
 ppr[.rm50 = 2 .rm60 = < .rm70 = F .rm80 = P .rm90 = Z, etc.]
 ppr
 pin        Your right margin SINGLE CHARACTER ----->  =$A
 ppr
 ppr
 ppradjusting squashticity
 b
 psx4 squashticity factor
 f<>.dbl<>.dbl>.rm+(x)<
 y?
 p
 e
 f<<>.rm-(x)><
 y?
 p
 pprimproving underline
 b
 f<<.ut><
 y?
d f<\< [esc]E<
 y?
 pgod1
 pgod2
d1 f<\<[esc]R <
 y?
 pgod
d2 b
 pprfix (.,)
 f<[esc]R .<[esc]R.<a
 p
 p
 f<[esc]R ,<[esc]R,<a
 p
 p
 f<.[esc]R<[esc]R.<a
 p
 p
 f<,[esc]R<[esc]R,<a
 p
 p
 f<. ><.><a
 p
```

PROGRAM 9-5 cont

```
 p
 pprfixing underline bug
 e
e u
 f<[esc]E<
 p
 pgoel
 pgoe2
el f<><>[esc]\<
 y?
 f<[esc]E<
 p
 pgoe
e2 p
 pprfix bold PS wheel
 b
 f<!<^<a
 p
 p
 b
 f<]<[esc]Z<a
 p
 p
 b
 f/</\/a
 p watch ut!
 p
 b
 f/[/</a
 p
 p
 b
 f/|/!/a
 p
 p
 b
 f/>/`/a
 p
 p
 b
 f/@/[esc]Y/a
 p
 p
 pprtightening spacing
 b
 f<>.dbl<
 p
 pgob
 pgob3
```

PROGRAM 9-5 cont

```
b h
 h
 f<>>><>[esc]U><
 y?
 pgob1
 pgob3
b1 f<>>><>>[esc]D><
 y?
 pgob2
 pgob3
b2 f<>>><
 p
 pgob
b3 p
 pprsetting body microjustify
 b
 f<>.dbl<
 p
 f<>.fj><[esc]X [esc]N [esc][tab]z [esc][tab]$A [esc]0 [e
             sc][tab]z [esc][tab][A][esc]/>.lj >[esc]M<A
 p
 pprshadowing titles
 b
 f<>.dbl<
 p
 pgoc
 pgoc1
c h
 h
 f<>.cj><>.cj>[esc]\[esc]X[esc]W[esc][Q][B][esc]%  <
 y?
 p
 f<><[esc]N[esc]P><
 y?
 h
 f<>.cj<
 p
 pgoc
c1 p
 pprfixing paragraph ends
 b
 f<>.dbl<
 p
 f<.><[esc]X.[esc]7[esc]/>[esc]M<a
 p
 b
 f<>.dbl<
 p
 f<^><[esc]X^[esc]7[esc]/>[esc]M<a
 p !=^ bold ps
```

PROGRAM 9-5 cont

```
 b
 f#?%#[esc]X?[esc]7[esc]/%[esc]M#a
 p
 b
 f<`><[esc]X`[esc]7[esc]/>[esc]M<a
 p >=` bold ps
 pin[G][G][G]--- detail work filename? ---> =$d
 psz+1  for wpl supervisor
 pas$d =$d
 pcs/ /$d/
 pgog
 pdo$d
g pqt
```

Gotchas: Pairs of brackets mean control commands. [esc] = escape key, [L] means "<ctrl>L", etc. Note that any isolated brackets really are isolated brackets.

The indented line in the "body justify" section is a continuation of the previous line and must be entered without intervening spaces or returns.

This WPL program assumes a Diablo 630 printer with an enhanced HPRO5 board having full word processing features. The program MUST be customized if any other printer is to be used. Certain features may not be available on other daisywheel configurations.

Be sure to follow the use rules in the text!

Follow those rules!

Once your document is formatted and dumped, there may still be some things you aren't happy with. That's where your detail files come in. You can do as much detail work as you feel you have to for any particular printing task.

Note that it is sometimes a good idea to add, change, or remove a word here or there to improve the balance and the overall appearance of your text on the final page. Don't be afraid to change the text to make it look better. The trick, of course, is to not change the meaning. Sometimes, a slightly wordy or slightly awkward structure will "read" better because it looks "cleaner" on the page. Hyphens can also be used to balance lines.

Another thing to watch for are the "widows" and "orphans" created when only a word or two ends up at the very top or very bottom body line on a page. There are lots of obvious solutions that include conditional form feeds, rewording, and changing ".pl" values on the fly.

You might not like the appearance of certain proportionally spaced letters when they end up beside one another. Type font design is an art, not a science. It is a real bear to get all the characters looking good beside one another.

Compromise is the watchword.

Fortunately, you can easily move two adjacent characters closer together by using the 1/120 inch little backspace under the "<open-apple>I" command. You can also stretch characters apart, but this is trickier. One way is to temporarily change the spacing table with a suitable adder. This is how the centered titles got spread out to

make up for shadow printing. Another way is to arrange the line wording or positioning so that the microjustifier will pull things slightly apart for you.

The proper name for pulling characters together is *kerning*. On hand-set lead type, kerning is done by notching the lower right side of one letter and the upper left side of the second one. The two type pieces then fit closer together, improving the balance between the visual spacing and the actual spacing.

Kerning can do wonders . . .

Examples of Kerning

1984	--->	1984
Thatcher	--->	Thatcher
VAPORLOCK	--->	VAPORLOCK

In the date, we have done a double little backspace to move the nine closer to the one. Dates look so bad the normal way, that you may always want to do this. The gap between the "a" and the "t" in the town name was closed with a single little backspace, as was the gap between the "t" and the "c". A double little backspace was also used between the V and the A in the tradename. Note how the bottom of the A is actually under the top of the V, yet you still get a visual balance.

Another nasty gotcha rears its ugly head: The AWIIe [F]ind and replace commands do *not* let you directly imbed backspaces or frontspaces. It seems that the "one-line" AWIIe entry prompter used by [F] refuses to accept backspaces, frontspaces, and real carriage returns. Interestingly enough, the one-liner does accept control commands directly, without any need for [V] imbedding. Note that this is significantly different than older versions of this program.

At any rate, neither you nor WPL can *directly* find, search, or replace an [H] or [esc] [H] backspace. To beat this, you have to search for the character pair where you want to insert the backspace. Then, you back up one with the left arrow key or use a single lowercase "h" as your WPL command. Then, pull the backspace or little backspace out of your glossary or off a disk text file.

Arrgh!

Naturally, you can play games like this forever. The trick is to do just enough detail work, word changing, and kerning to give you acceptable results for what you are trying to accomplish.

As to "perfect" print quality you can't get there from here.

Short of gravure.

WHAT NEXT?

Either of the fully automatic formatters and any detail files can themselves be controlled by another "master control" supervisory WPL program. This is most useful for printing entire book chapters or working over several disk drives at once. To do this, use the $D string to hold the name of your detail file and bypass any user input. You can also increment the (z) counter in WPL to let one supervisory program reuse

the automatic formatter many times over. The detail file should then rerun the supervisory program.

To handle this last trick, your main WPL program should check its (z) numeric variable to find out which text file it is working with. The supervisor sets (z) to one and gets a file, formats it, details it, and prints it. The supervisor is then rerun. (z) is checked. This time, (z) is now *two* since it got incremented. So, the second text file is given the royal treatment. Continue this for the entire document over as many files and drives that you care to.

What you have done is used entire WPL programs to act as subroutines for each other, keying on a (z) "program counter." The advantage of this over regular WPL subroutines is that your total program can now be vastly longer than the 2048 character WPL single-program limit.

Should you need footnotes, you can shorten your WPL programs to 1024 or fewer characters, and then chain them in the usual way.

There is supposedly a way to pick up a limited 630 hyphenation capability since the *Diablo* can be trained to automatically carriage return on any hyphen within five

SEEDS and STEMS

65C02 Upgrade

There is a brand new member of the 6502 family called the 65C02. The "C" stands for CMOS. These chips are available from Western Design Center, GTE, Rockwell, or Synertek.

You can directly replace the 6502 on a IIe with a 65C02. The only immediate benefit will be a cooler Apple IIe with more supply power available for plug-ins. Unfortunately, only some of the 65C02s will work on older Apples, due to timing changes.

For the future, though, you get many more op codes that do many new and different things. Any new programs can be noticeably shorter and faster.

Even better yet, all of the "illegal" op codes now default to NOPs. Which means you can add some simple external hardware to microprogram your 65C02 into a virtually unlimited number of op codes and address modes. A 50X speedup in graphics animation is one obvious possibility.

More exciting still is a brand new 16-bit 6502 upgrade that, believe it or not, is completely pin compatible with the original 6502. Contact the Western Design Center for details on this beauty.

spaces of the right margin. I haven't looked much further into this, but it could be just what you need.

Lastly, one of the big myths of computerdom is that you cannot do decent graphics on a daisywheel printer. As Fig. 9-6 shows us, this myth is an outright lie. In fact, some graphics can be done much better by daisywheel printers than by dot-matrix printers. We'll save more details on this for a future enhancement.

Now it is your turn.

What can you do with all these exciting new word processing features that will lead you toward superb print quality? Use the response card or the hotline to close the loop, letting us know what you come up with.

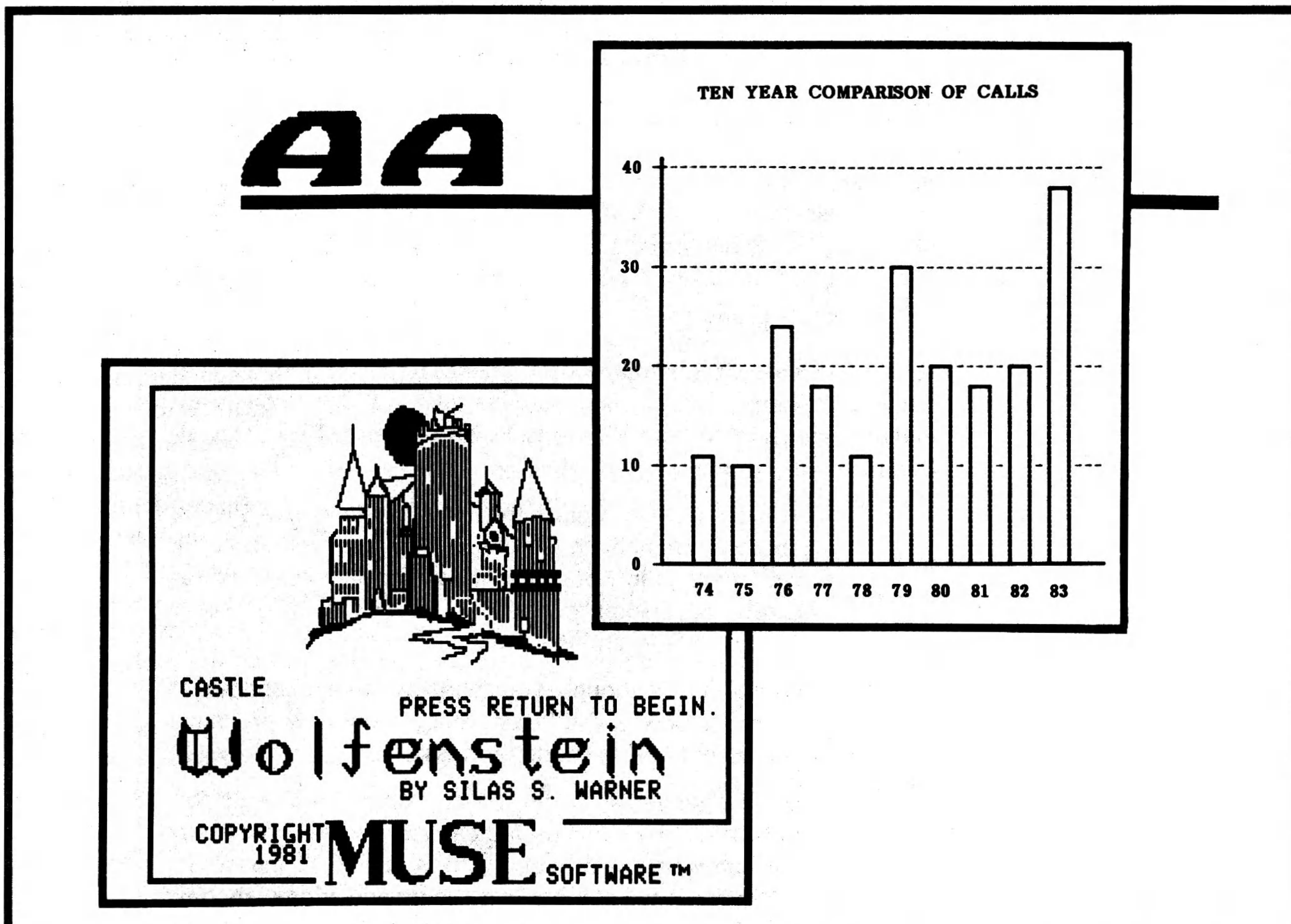


FIGURE 9-6. High quality graphics are easily done using daisywheel printers, and HIRES dumps can be done from within a slightly modified Apple Writer IIe.

The following programs are included on the companion diskette to this volume:

DGLOSS
EGLOSS
AWIIE NULLIFIER
AWIIE STRETCHIFIER
WPL.FORMAT DIABLO 630
WPL.DETAIL VAPORLOCK
WPL.DETAIL E9
WPL.LISTER
WPL.CAMERA READY

Sixteen additional disk sides of Apple Writer tools and tutorials are also available.

See the response cards in the back of the book for more details.

This enhancement works only on the Apple IIe, although EEPROM programming details shown here are good for use anywhere.

Enhancement 10

ABSOLUTE "OLD MONITOR" RESET FOR THE IIe

There's no need to put up with the inanities of the Apple IIe's reboot process. You can get back into total control with this simple, $6.00 monitor modification. Included are full details on programming new EPROMs on old EPROM burners.

ABSOLUTE "OLD MONITOR" RESET FOR THE IIe

Once upon a time, in a magic kingdom far away, there lived a truly wondrous automobile.

It was the first "no excuses, no apologies" automobile ever available and was very popular among the princes and the populace alike. Alas, this otherwise stupendous machine had a single and very serious flaw. A flaw so insane and so incredibly stupid that it could only have been placed there by a demented and wicked witch.

For, you see, this wondrous automobile had a large pedal on the floor that was plainly marked "BRAKE." Drivers of this wondrous automobile expected and assumed that, when this "BRAKE" pedal was pressed, the automobile would be brought to a swift and safe stop, without harm to the driver, passengers, or any cargo.

But this was nought to be. Forsooth, the "BRAKE" pedal was really a magical pedal under a horrible spell. If the "BRAKE" pedal was pressed by itself, the automobile, the driver, and all contents got magically and instantly whisked back to the carriage house from whence the trip began.

If a driver was foolish enough to press the "BRAKE" pedal at the same time he turned on the windshield wipers, the wondrous automobile did, in fact, come to an immediate stop. But, alas and alack, the immediate stop was so sudden and so violent that it destroyed the driver, passengers, and all contents of the magical vehicle.

Well, not really destroyed. For you see, all that really happened is that a pair of holes, five "urflogs" in diameter, got neatly punched *completely through* the driver, any passengers, and all else that happenstance had placed inside the wondrous machine.

When the grand vizeers were asked why the "BRAKE" pedal was not really a "BRAKE" pedal, but instead an evil and demonic device, they offered two reasons.

Some vizeers said that the multitudinous makers of hood ornaments and glove compartment door hinges did not want the drivers bringing their vehicles to a quick and safe stop, since the hood ornaments and glove compartment door hinges could then — horror of unspeakable horrors — actually be *inspected* and possibly *modified* by the driver.

Others said that there must be some protection to keep the driver from inadvertently and unintentionally pressing the "BRAKE" pedal if he did not, in fact, really want to bring his wondrous automobile to a swift and safe stop. And, indeed, a much older model of the same wondrous automobile, did have its "BRAKE" pedal situated where it could easily be mistaken for the horn ring.

Lo and behold, a certain driver of the wondrous automobile finally decided he had more than enough of this male bovine excreta.

He pulled out the old "BRAKE" pedal by its roots and threw it away. Then, he replaced the magical "BRAKE" pedal with a real one that was able to swiftly and controllably bring his wondrous automobile to a safe and sure stop. Having done so, that driver grabbed the nearest handy princess, drove off into the sunset, and lived happily ever after.

A fable you say? Only perhaps.

The Apple IIe monitor has a fatal flaw. Not in the code itself, but in the credibility of what a system monitor is supposed to do. Fortunately, the IIe monitor is resident in chips that can be swapped for stock 64K EPROMs. You can easily and swiftly change the Apple IIe firmware to do things the way you would like them done, rather than the ways those who think they are in power choose to dictate unto you.

Let's see what is involved in burning your own 64K EPROMs, using them as IIe monitor substitutes, and then providing your own "old" monitor absolute reset.

The code we will show you for an improved monitor does everything the original IIe monitor does, except for two crucial differences. First, there is no "hole blasting" done

on a cold restart. Secondly, if you keep your finger on the open-apple key for a minimum of four seconds *after* a cold reboot, you will be dropped *directly* in the "old" monitor, awaiting your machine language commands.

We will show you details here for the "old" IIe ROM. Contact the helpline for free information on the IIc ROM or the "new" IIe ROM replacement. You can tell an "old" IIe ROM by a 6502 (rather than a 65C02) in a stock machine, and its refusal to accept lowercase Applesoft commands. "Old" IIe ROMs were shipped up to January of 1985. What we show you here will *not* directly work on a IIc or a "new" IIe.

We'll call this new monitor a KREBF monitor, named after the magic spell that "repairs willful damage" in Infocom's *Enchanter* adventure. Actually, we won't repair any willful damage. We just won't do any damage in the first place.

Beyond this simple and special monitor change, you can now custom modify Applesoft, or, for that matter, provide your own entire operating system for special Apple IIe uses. You can also use a custom pair of monitor EPROMs to do most of what a "snapshot" card will do, at a tiny fraction of the cost.

Legally, I can not just "give" you an EPROM to plug into your Apple because of the copyright hassles involved in the Apple firmware. Instead, I will show you a painless and fully automatic process for converting your own firmware into a form useful for EPROM programming. I will also give you the full *new* source code patch and detailed instructions that will return absolute control of your IIe back where it rightly belonged in the first place. I will also show you one EPROM programming service.

The rest is up to you.

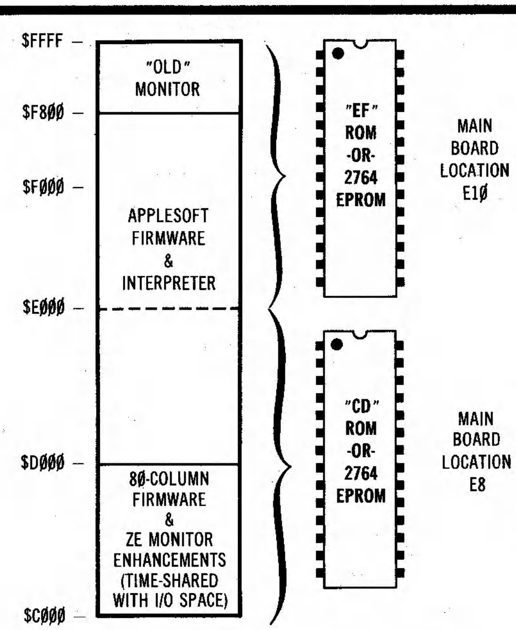


FIGURE 10-1. Apple IIe firmware is held in a pair of 8K × 8 ROMs. Standard 2764 EPROMS may be directly substituted, letting you easily customize your IIe.

ABOUT IIe FIRMWARE

The stock Apple IIe has 16K of ROM-resident firmware that sits between $C000 and $FFFF. As Fig. 10-1 shows us, this firmware is held in two 64K read only memories. One of these is called the "CD" memory and sits in main board location E8, while the second is called the "EF" memory and sits in board location E10.

The traditional monitor area on older Apple II's needed only the 2K space from $F800–$FFFF. This area is still used as a small part of the IIe monitor. To provide for the many new IIe features, expansion hooks have been added to also allow use of the $C000–CFFF memory area.

Since $C000–CFFF is in the I/O, or input/output, area, special soft switches are used to pick either "normal" I/O or "monitor" use of this address range. Thus, anything that wants use of the "new" IIe monitor area must first turn off the I/O and then turn on the "CD" firmware ROM. When you are finished using the new monitor area, the "CD" ROM must be turned back off and the I/O must be reactivated.

There are four soft switches involved. One pair handles only the memory area from $C300–C3FF and is used to make the 80-column firmware look like it is sitting in slot 3 of the I/O space. The second pair of soft switches is used to switch everything else and is called the $CX00 switches. The "X" here can be a 1, 2, 4, 5, 6, or 7.

The top half of the CD memory and the bottom three-quarters of the EF memory hold the Applesoft firmware. This code is apparently unchanged from earlier models in the "old" IIe firmware.

Older Apples used 16K read-only memories, or ROMs, that were not quite compatible with industry standard 2716 EPROMs. One enable pin was active high on the ROM, compared to an active low state needed by the EPROM. While you could just change some jumpers around to do some short-sighted 2716 replacements, to do the job right, an inverter and a small plug-in card was needed.

Without the inverter, certain plug-in cards could cause memory contention and hang the machine. Quite a few articles on adding EPROMs to older Apples ignore this key point, leaving you with a potential time bomb on your hands.

Thus, swapping for 2716 EPROMs did not get done much on older Apples because of the hassles involved.

Very fortunately, the 64K read-only memory firmware used in the IIe is directly and exactly compatible with the standard 2764 EPROM. So, to customize things any way you like, you simply swap chips . . .

> The "CD" or "EF" chips in a IIe may be directly replaced by 2764 EPROMs.

In case you have tuned into the microcomputer revolution late, a 2764 is a special "read-mostly" memory that you can custom program and reprogram yourself. You buy these for around $6.00 from ads in the back of any decent computer magazine. You erase any old memory contents by using a special ultraviolet lamp. You reprogram the memory to suit yourself by using either a programming card for the Apple or a stand-alone EPROM programmer.

EPROM programmers are readily available at any hacker's club if you do not already own one. We will shortly see a sneaky way to program the 2764 on older burners that may not be directly able to handle such a large EPROM.

One commercial EPROM burning service is E-TECH SERVICES, Box 2061, Everett, WA 98203 (206) 337-2370. They are specifically set up to burn monitor EPROM.

To customize your IIe to suit yourself, all you have to do is burn one or two 2764 EPROMS and then swap them for the CD or EF memory they are to replace. That quick and that simple.

There is one minor gotcha though . . .

> The 2764 EPROM used MUST be an Intel or Hitachi brand with 28 pins and an access time of 250 nanoseconds or less.

It seems there are two kinds of 2764 EPROMs kicking around these days, real ones and fake ones. Real 2764s don't eat 24-pin sockets.

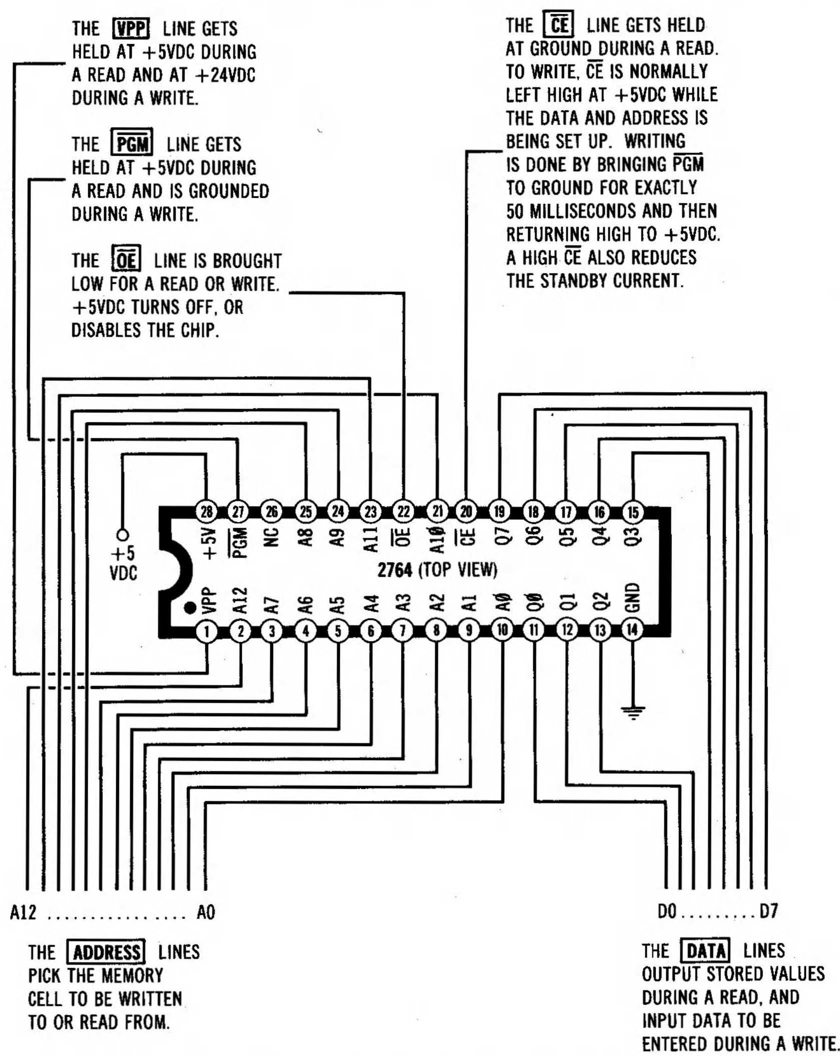


FIGURE 10-2. How a 2764 is used.

Real 2764s are made by Intel, Hitachi, and several other mainstream suppliers. These always come in a 28-pin package, and are the *only* type of 2764 usable as a direct Apple monitor replacement. Both Motorola and Texas Instruments have their own imitation versions of fake 2764s that come in 24-pin packages. These are not compatible with anything anywhere, not even with each other.

Be sure to use a real 28-pin Intel or Hitachi 2764!

ABOUT 64K EPROMS

Fig. 10-2 shows us the pinouts for a real 2764, along with some use details.

To analyze any memory chip, break the package leads down into four groups of supply, address, data, and control lines. Then analyze each group.

Only a single +5-volt supply and ground is needed for normal reading of the 2764 EPROM. Supply current is typically around 150 milliamperes in the read mode.

The 2764 is a 64K memory that is organized as 8K by 8. This means that there are 65,536 bit locations that are programmable to a one or a zero. These bit locations are arranged into word bytes of 8 bits each. There are 8192 different 8-bit words.

The address lines pick which word is to be written to or read from. Thirteen address lines are needed, since 2 ↑ 13 = 8192. To select a particular 8-bit word, the correct binary pattern of ones and zeros is placed on the 13 address lines. Internal address decoding inside the chip then picks the correct byte for reading or writing.

The address lines always input *from* the microcomputer *to* the 2764.

Turning to our third group of lines, there are eight data lines. These data lines are used to route the contents of the addressed byte to the microcomputer during a read. The same lines are used to send the data to be written into the EPROM during a write.

Thus, the data lines *input to* the EPROM during a write, and *output from* the EPROM during a read.

It turns out there are five possible activities an EPROM can be up to . . .

EPROM Activities	
Erase	— Clears the entire memory when exposed to strong ultraviolet light.
Program	— Writes a single byte into memory.
Verify	— Checks a byte just written during programming.
Read	— Outputs a previously programmed data value.
Standby	— Does nothing, allowing other data bus uses.

These activities are handled by four control lines, call VPP, $\overline{\text{PGM}}$, $\overline{\text{OE}}$, and $\overline{\text{CE}}$.

To use an EPROM, you first have to program it. Before you program it, you have to erase anything old that was previously stored in it. Erasure forces all of the data bits to ones, and all bytes to $FF's. This erasing is done with a special ultraviolet lamp.

Alternately, you can leave the chip out in bright sunlight for a week.

Unlike newer EPROMs, the 2764 has no way to "unprogram" a single byte. You erase the entire chip at once.

The VPP, or *programming voltage,* line provides normal +5 volts dc for standby and reading. To program, this line must be brought positive to +21 to +24 volts. Earlier EPROMs did not have this programming voltage on a separate pin, which complicated things. On some micro systems, it is a simple matter to in-socket program or alter a 2764. Unfortunately, the stock Apple IIe does not have this capability.

At any rate, leave VPP at +5 to do anything but program. Raise it to +24 to program.

Incidentally, most 2764s can use either a +21- or a +24-volt VPP programming voltage. Some older 2732As and a few 2764As demand a maximum of +21 volts while being programmed and will self-destruct on the traditional +24 volts. Check the data sheet for the exact brand you are using if there is any doubt.

The $\overline{\text{PGM}}$, or *program* line is a normal logic control signal. This one is grounded to program and is set to +5 for everything else.

The $\overline{\text{OE}}$, or *output enable* line is brought low for a read or a write. A high $\overline{\text{OE}}$ line disconnects the outputs of the EPROM, but leaves the chip addressed and fully powered.

The $\overline{\text{CE}}$, or *chip enable* is used to turn the entire EPROM on or off. During a read, $\overline{\text{CE}}$ must be brought low. During a write, $\overline{\text{CE}}$ is held high until an address is selected and data is input. Then $\overline{\text{CE}}$ is brought low for exactly 50 milliseconds to blast the data value into the EPROM. $\overline{\text{CE}}$ is then returned high while the address changes for the next byte. A high $\overline{\text{CE}}$ also greatly reduces the standby supply power needed.

Apple chose to use the $\overline{\text{OE}}$ line as an alternate chip enable, since it is faster. They were not in the least worried about saving any supply power, since they already had problems meeting the *minimum* power supply current drain during the IIe redesign.

Apple also permanently grounded the $\overline{\text{CE}}$ line and permanently held the $\overline{\text{PGM}}$ line at +5, forcing any 2764 plugged into the main board into a read-only mode.

Thus, unless you want to do some major board carving, "in situ" programming of a 2764 EPROM in an Apple IIe is not normally feasible. Instead, you have to use an EPROM programmer or programming card and then swap out the chips.

Let's summarize these control lines . . .

2764 Control Lines	
VPP (pin 1)	— Raise it to +24 vdc for programming, but hold it at +5 for anything else.
$\overline{\text{PGM}}$ (pin 27)	—Ground it to program, but hold it at +5 for everything else.
$\overline{\text{OE}}$ (pin 22)	— Ground it all the time, unless you are using it as a faster read chip enable.
$\overline{\text{CE}}$ (pin 20)	— Ground it to read. To program, hold it high, but bring low for 50 milliseconds after address and data are stable.

Reviewing, we have four control lines. VPP provides the special +21 to +24 volt supply power needed during programming. $\overline{PGM}$ inhibits programming unless it is grounded. $\overline{OE}$ is normally grounded, unless you are using it to speed up your system access. $\overline{CE}$ is grounded for a read and held high for everything else. To blast one byte, you stabilize your address and data lines and then bring $\overline{CE}$ low for 50 milliseconds.

We saw how we also had eight data lines that output from the EPROM during a read and that input to the EPROM during a write. There are 13 address lines that pick which of the 8K locations are to be written to or read from. Finally, there is the normal +5 supply line, which is all you need for read or standby operation.

For more information on this and other memory chips, check into *Don Lancaster's Micro Cookbooks* (Sams Cat. Nos. 21828 and 21829).

Except for one little detail, we are almost ready to blast some bits . . .

PROGRAMMING A 2764 ON AN OLD BURNER

Today, the 2764 costs around $6.00 and it will almost certainly drop further in price when the larger 27128s and 27256s reach jellybean status.

Many of the older EPROM burners cannot directly handle a 2764, and you won't find too many newer models available yet that do. Some of the nasties involved in physically upgrading an older burner include going from a 24 pin to a 28- pin socket, providing that thirteenth address line, getting a PGM signal to the socket, or figuring out how to stuff 8K worth of data into the 4K I/O space normally available on an Apple IIe.

SEEDS and STEMS

NEVER Write to a Game Diskette!

ALL game diskettes WILL eventually fail. It may be because the oxide is literally ground off the diskette. Or too much peanut butter and jelly gets on the disk surface. Or a drive speed error. Or dirty or loose contacts. Somehow, some way, the game disk WILL eventually fail.

If a game diskette of yours fails later than an average game diskette, then your customers will be very happy with you. If, instead, early failures are common, you get a very bad reputation very fast. No matter that the drive is slow, cables loose, and fingers sticky. You get blamed.

Which says that diskettes that are used to load only will last much longer than diskettes that are continually used during a game session. Even more importantly, if you EVER write to a game diskette, you WILL dramatically shorten the average life. Even to only save a high score. The reason is that you can write the wrong thing to the wrong place, plowing the otherwise usable works.

Hence the above rule.

We can bypass all these hassles by adding a fairly simple and easy-to-use adapter to an older EPROM burner. This adapter will make the 2764 to be programmed look like a sequential pair of older 2732s. If you can program a 2732, as most older EPROM burners can, you can easily program a 2764 with this adapter.

You'll find a two position slide switch on the adapter. Put the switch in the low position to program the low 4K of your 2764 as if it was a 2732. Then flip the switch to the high position so you can program the high 4K of your 2764 as if it was a separate 2732.

That simple.

Once again, if you do not already own an older EPROM burner, you should be able to borrow one at most any school or club. EPROM burner plug-ins are also widely available in the $50–$150 price range.

Our adapter is intended for use with the *MPC* "ap-ep" EPROM burner peripheral card for the Apple II or IIe. Certain details might change for other burners.

At any rate, here are the parts you will need . . .

Parts Needed for a 2764 EPROM Burner Adapter

1 — 28-pin machined contact DIP socket
1 — 24-pin machined contact DIP socket
1 — 11-pin machined contact DIP strip
1 — 7-pin machined contact DIP strip
1 — 3-pin machined contact DIP strip

2 — bare machined contact DIP pins
1 — "extra" DIP socket for heatsink
1 — miniature SPDT slide switch
1 — 1N4001 silicon power diode
1 — mini-grabber test clip

1 — ¼-watt resistor, any value
1 — 4" white No. 22 stranded wire
1 — 2" bare No. 24 solid wire
1 — 2" red No. 24 solid wire
2 — 2" green No. 24 solid wire
1 — 8" piece of electronic solder

optional — super glue, epoxy, or silicon rubber

These parts should be readily available anywhere, although you MUST be certain to use the type of premium *machined contact* DIP socket that can safely be plugged into one another. Do NOT use regular el-cheapo sockets.

You also MUST use a slide switch, rather than a toggle switch. Toggle switches work "backwards" and will scramble the 2764.

A schematic of the adapter is shown in Fig. 10-3.

We'll skip a list of tools, for all you will need are the usual small soldering iron, dikes, needle nose pliers, wire stripper, and some small support vise.

Note that this adapter and these tools are *only* needed if you are having trouble burning a 2764 EPROM on an older EPROM burner. All you need to do the actual upgrade on your IIe is a progammed 2764 and a simple chip swap.

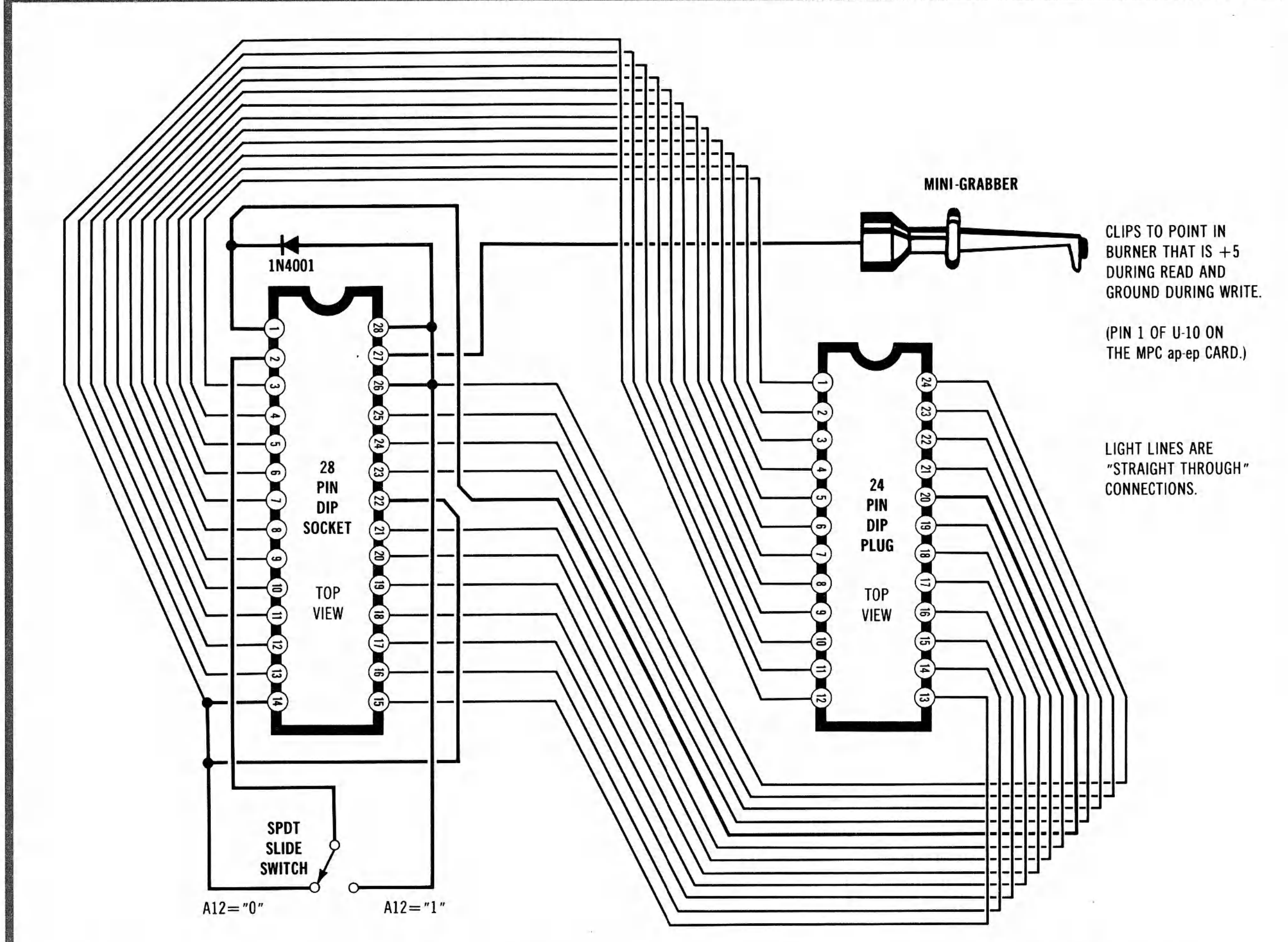


FIGURE 10-3. This adapter will let you program a 2764 EPROM in an older EPROM burner set up for a 2732.

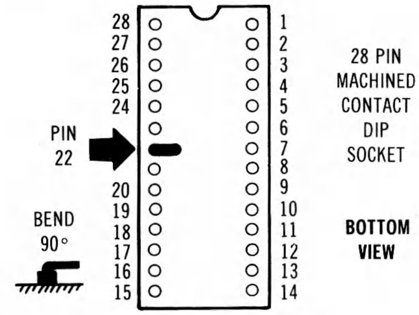
Fig. 10-4 gives you complete construction details for the adapter. It is very easy to build, and should take all of one-half hour. Be sure to use an "extra" machined contact DIP socket or socket strip when you are soldering anything. This will keep the pins properly spaced and aligned should the plastic soften. Plug this "extra" socket into the cool ends of the pins being soldered so it straddles any pins to be heated.

If you use the optional epoxy or super glue, be careful not to get any gunk inside the slide switch. Carefully test the switch after the glue sets.

NOTE: This adapter is needed ONLY if you must burn a 2764 EPROM on an older 2732 burner. You do NOT need this adapter to replace a IIe monitor ROM with a 2764 EPROM.

1. Turn the 28 pin machined contact DIP socket over and identify pin 22 by inking the plastic. Then carefully
( ) bend pin 22 towards the center as shown.

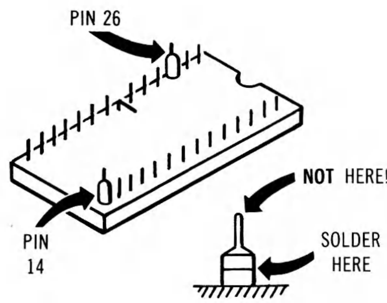
   Be sure to use the long 28 pin socket, and NOT the short 24 pin one for the steps that follow. Also be sure to work from the socket BOTTOM SIDE UP. Note that the pins count backwards from usual when you do this.



2. Push a single bare machined contact socket pin onto pin 14 as shown. Then push a second socket pin onto
( ) pin 26, also as shown.

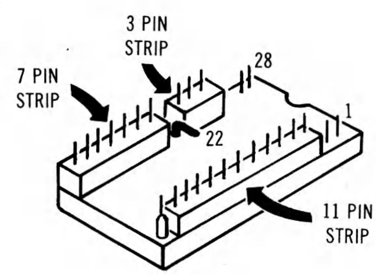
   NOTE: in any soldering steps that follow, use the "ex-tra" machined contact DIP strip or socket as a heat sink by clipping it onto the cool end of the pins being soldered. This will keep the pins aligned should the plastic soften.

   Carefully solder the bare pins to socket pins 14 and 26. Use very little solder. DO NOT GET ANY SOLDER ON THE PIN TIPS!



(3) Push an 11 pin machined contact DIP strip on pins 3–13 as shown.
( )
   Push a 7 pin machined contact DIP strip onto pins 15–21 as shown.

   Push a 3 pin machined contact DIP strip onto pins 23–25 as shown.



4. Carefully roughen one side of the SPDT slide switch and the bottom of the 28 pin DIP socket between pins
( ) 14 and 15. Use very fine sandpaper, steel wool, or an ink eraser.

   Glue the switch to the 28 pin DIP socket as shown, us-ing a drop of superglue or VERY LITTLE epoxy.

   DO NOT GET ANY GLUE INSIDE THE SWITCH!

   Let the adapter dry overnight. Then, verify that the switch still works.

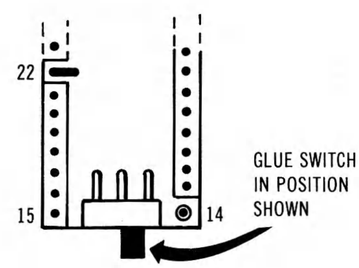
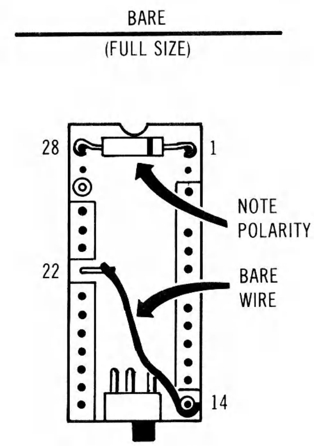


FIGURE 10-4. **Building your 2764 adapter.**

5. Prepare a two inch length of bare #24 wire. Connect this wire first to pin 14 of the 28 pin socket, then to
( ) the nearest pin on the SPDT slide switch, and finally to bent pin 22.

   Solder all three connections. Use the "extra" DIP strip or socket as a heatsink, moving it as needed.

   Cut off any extra wire remaining.

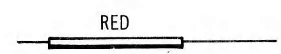


6. Connect the 1N4001 silicon diode from pin 1 to pin 28, of the 28 pin DIP socket, making sure that the cathode
( ) bar of the diode goes to pin 1. Make mechanically secure connections, but do not solder them just yet.

(SEE ABOVE FIGURE)

7. Take a 2-inch piece of red 24 solid insulated wire and strip ¼ inch from one end and ¾ inch from the other.
( )
   Solder the ¼ inch end of this wire to the far unused pin on the SPDT slide switch. Loop the ¾ inch end of this wire around pin 26 and then to the diode lead coming from pin 28.

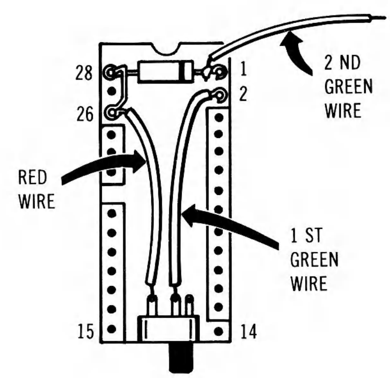
   Using the "extra" DIP socket as a heatsink, solder pin 26 to the red wire and pin 28 to the diode lead and the wire. Cut off any extra wire.



8. Prepare two 2-inch green 24 insulated wire by stripping ¼ inch off each end. Solder one end of one green
( ) wire to the center SPDT switch terminal. Solder the other end of this green wire to pin 2 as shown. Use the "extra" DIP socket as a heatsink.

9. Solder one end of the second green wire to pin 1 of the 28 pin DIP socket and the cathode (bar) lead of the
( ) power diode. Use the "extra" DIP socket as a heat-sink.

   Do not connect the other end of this wire just yet.



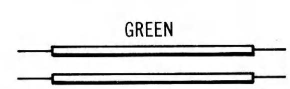
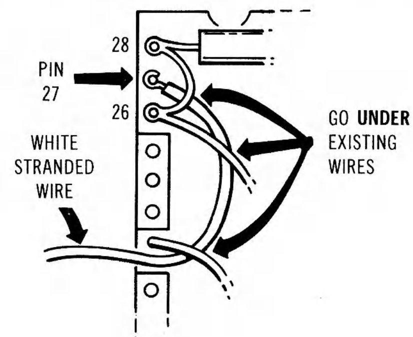


FIGURE 10-4. cont

10. Prepare a 4 inch white 22 stranded wire by stripping ¼ inch off each end. Route this wire UNDER the bare wire at pin 22 and UNDER the wire between pins 26 and 28 of the 28 pin DIP socket. Then solder one end of this white wire to pin 27. Use the "extra" DIP socket as a heatsink. ( )



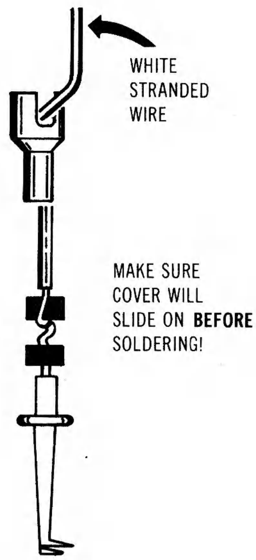
11. If you care to, place a small blob of silicon rubber sealant over all the wires and the back of the switch. ( )

    This will act as a strain relief and further support the switch. Do not get any glop inside the switch.

12. Study the grabber to see how the wire gets attached. Slide the cover of the grabber over the white wire, and then do a trail assembly, WITHOUT SOLDERING to make sure the grabber will go together properly. ( )

    Then solder the grabber to the wire using a minimum of solder.

    Slide the grabber back together. Then test the grabber to be sure it works properly.



13. Take the remaining 24 pin machined contact DIP socket and identify pin 20 by inking the plastic. ( )

    Push the bare end of the flying green wire into the socket (TOP) end of pin 20 on the 24 pin socket.

    Align the sockets so that pin 14 of the 28 pin socket is above pin 12 of the 24 pin socket, and that the notches are at the same end, but staggered from each other.

    Solder the green wire to pin 20 on the 24 pin socket. Use the "extra" DIP socket as a heatsink. Be sure to solder to the TOP side of the 24 pin socket.

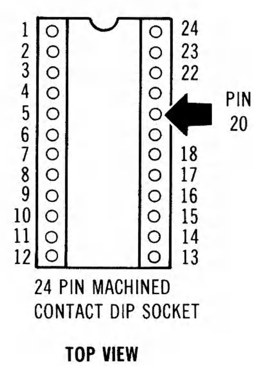
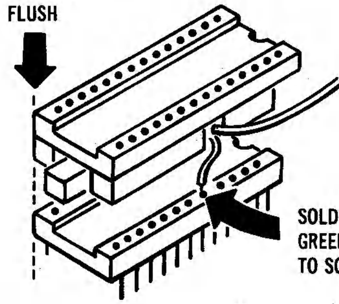


FIGURE 10-4. cont

14. Snap the two sockets together, being sure that the switch end is flush and the notch ends are staggered.
( )
Verify that Pin 3 of the 28 pin socket goes to pin 1 of the 24 pin socket.

This completes the adapter.

If you are going to program lots of 2764's, you might want to plug or solder a 28 pin ZIF socket into your adapter. These are fairly expensive.



15. Study the schematic of your EPROM burner and find a point that is grounded during WRITE and at +5 during
( ) READ. Place a small wire hook or loop at this point in the burner circuitry.

On the MPC ap-ep EPROM card, carefully tin the very top of pin 1 of U10, and then add a small wire loop as shown in the inset to Figure 10-5. Test grab this loop with the grabber.

FIGURE 10-4. cont

You might later want to add a 28-pin ZIF, or zero insertion force, socket to the top of the regular socket. Depending on the pins on the ZIF, this may plug in or may have to be soldered. ZIF sockets are usually rather expensive.

Fig. 10-5 shows you how to use the adapter on the MPC "ap-ep" card. A very slight modification to the card itself is needed. You have to add a small wire hook to the top of pin 1 of integrated circuit U10. The easiest way to do this is to tin the top of the pin with a very small amount of solder. Then bend a hook in the end of a resistor lead. Reflow solder the resistor lead in place. Finally, cut away all of the resistor except enough lead to form a small loop. Then, throw away the resistor, or reuse it elsewhere.

If you are using a different card or stand-alone burner, make sure you find a place that is at ground during writing and at +5 during reading. Add a small wire hook to this point so the grabber can access this signal. Also be sure that pin 1 will be at +5 volts during read and at +21 or +24 volts during programming.

One detail. Note that the 2732 requires a VPP signal of ground for a read and +21 or +24 volts for a write. The 2764 instead requires a VPP signal of +5 volts for a read and +21 or +24 volts for a write. This read supply difference is the purpose of the diode between adapter pins 1 and 28. If you are using an oddball EPROM burner, make sure that no "hard" ground shorts this diode or a supply line.

To install the adapter, put it in the existing 24-pin ZIF socket so that the switch points away from the ZIF handle. You might have to jiggle the handle slightly, center the adapter, and then slide the locking handle home, as there is a very slight "negative clearance" between the open handle and the adapter. Then glomp the grabber onto the wire hook.

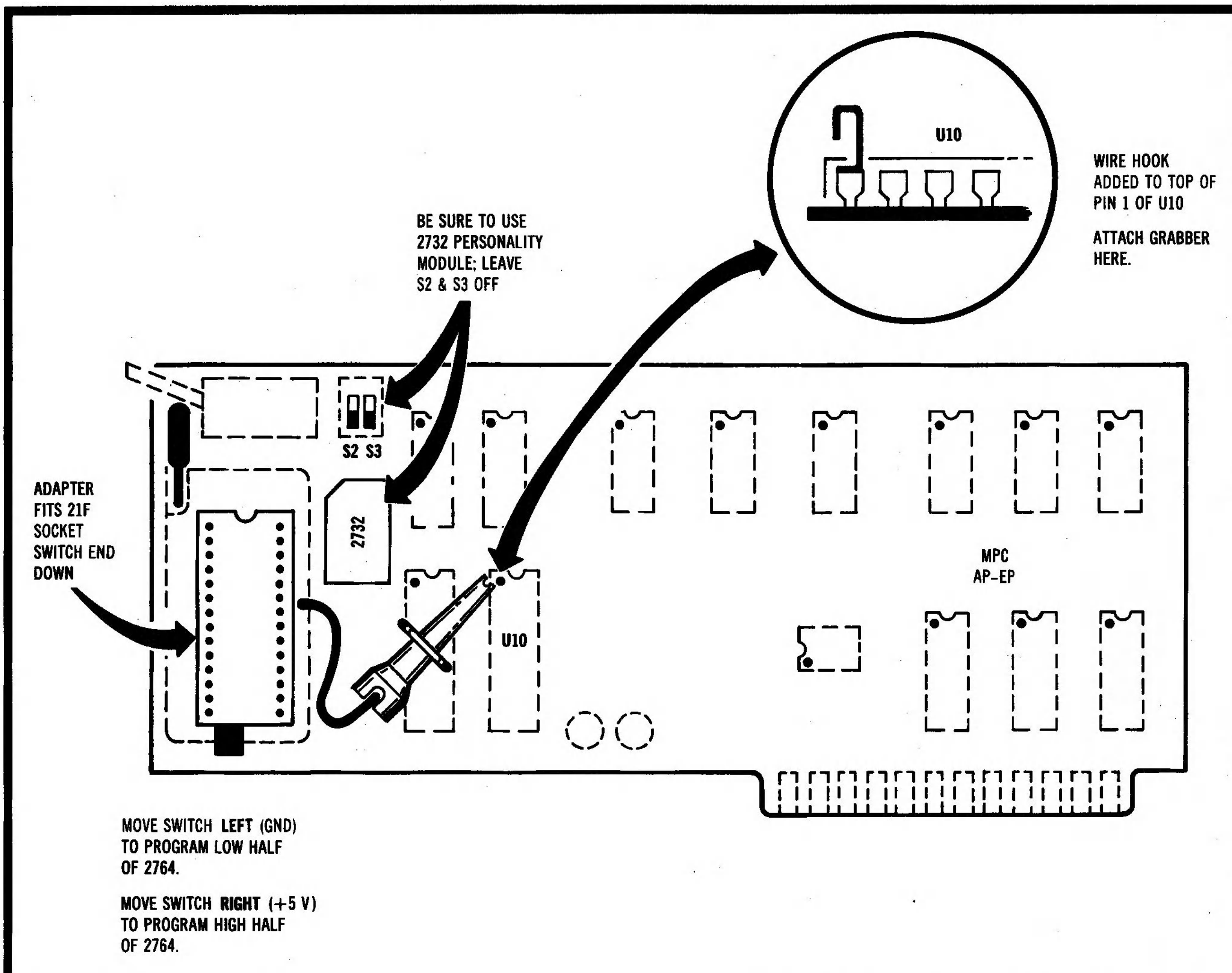


FIGURE 10-5. Using the 2764 adapter on the MPC "ap-ep" burner card.

Two gotchas . . .

> Be CERTAIN that the EPROM burner or card is configured for a 2732 burn!
>
> NEVER try connecting or disconnecting the grabber with a 2764 in place!

As a special note to MPC "ap-ep" users, also be sure both S2 and S3 are in their OFF position, pointing towards the bottom of the card.

Here are the rules on the slide switch use . . .

> The slide switch points to GROUND (or pin 14) for a low address A12.
>
> The slide switch points to +5 VOLTS (or pin 28) for a high address A12.
>
> Use the LOW switch position for a "C" or "E" monitor burn.
>
> Use the HIGH switch position for a "D" or "F" monitor burn.

You might like to mark a "0" on the left switch position and a "1" on the right switch position.

Well, we now have a way to substitute for a IIe monitor ROM. And we now have a way to program a 2764 on an older burner. All we need now is a way of . . .

CAPTURING THE IIe MONITOR

Catching the "D", "E", and "F" segments of the monitor ROM is rather trivial. In fact, some burner card software will let you simply move their buffer directly to these locations.

But the $C000–CFFF monitor capture is a bit tricky and nonobvious.

So, Program 10-1 shows you an Applesoft program called SNATCHMON. What SNATCHMON does is grab four 4K segments of the monitor. It places these on disk under the names of IIEMON.C, IIEMON.D., IIEMON.E., and IIEMON.F.

You then modify these four segments as needed for your custom monitor.

These are saved to disk intending to be BLOADed to a buffer at $8000–8FFF. You can relocate these as needed if your EPROM burner card requires a different buffer area. If you have a stand-alone burner, you can use a modem program to send these binary files out as serial data, and then use them as needed.

This Applesoft program automatically captures your existing Apple IIe monitor firmware and converts it into files convenient for customizing and EPROM burning.

SNATCHMON is suprisingly fast, since all of the actual moves are done using the monitor "M" command. Suitable POKEs activate each move as needed. It is absolutely essential that the Y register be zeroed before a monitor move. To do this, any SNATCHMON move CALL first goes to a five byte machine program at $7FFB that clears the Y register and then jumps to the actual move routine at $FE2C.

Locations $3C (low) and $3D (high) hold the move starting source address. Locations $3E (low) and $3F (high) hold the move ending address. Finally, locations $42 (low) and $43 (high) hold the starting destination address. After these six locations are POKEd with the right values, the move can be made.

If you're having trouble converting Applesoft decimal into machine hex, check into the *Hexadecimal Chronicles* (Sams Cat. No. 21802) for instant conversions.

To capture the $C100-CFFF monitor as IIEMON.C, you have to flip two pairs of soft switches just right. The magic switches involve $C006, $C007, $C00A, and $C00B. Soft switch $C007 turns ON the monitor for the $CX00 access, while $C006 turns ON

PROGRAM 10-1 Listing of SNATCHMON.

```
10   REM
18   REM ***********************
20   REM *                     *
22   REM *     "SNATCHMON"     *
24   REM * IIE MONITOR GRABBER *
26   REM *  FOR EPROM BURNERS  *
28   REM *                     *
30   REM *     VERSION 1.0     *
32   REM *.....................*
34   REM *                     *
36   REM *  COPYRIGHT 1984 BY  *
38   REM *  DON LANCASTER AND  *
40   REM *  SYNERGETICS,  BOX  *
42   REM *  1300 THATCHER AZ.  *
44   REM * 85552. 602-428-4073 *
46   REM *                     *
48   REM *   ALL  COMMERCIAL   *
50   REM *   RIGHTS RESERVED   *
52   REM *                     *
54   REM *********************** [J]
 [J]

80   REM  THIS PROGRAM "CAPTURES"
82   REM  THE APPLE IIE MONITOR
84   REM  INTO FOUR 4K WORKFILES
86   REM  FOR USE WITH AN EPROM
88   REM  BURNER WHOSE WORK FILES
90   REM  BEGIN AT HEX $8000. [J]

99   REM  [J]
        ..................... [J]
100   TEXT : HOME : CLEAR : GOSUB 2000: REM  GET TUTORIAL
110   VTAB 12: HTAB 7: PRINT "MONITOR SNATCH IN PROGRESS"
120   PRINT : FLASH : HTAB 14: PRINT "PLEASE WAIT": NORMAL
199   REM  [J]
         ..................... [J]

200   REM : $C100-CFFF GRAB [J]

210   POKE 32763,160: POKE 32764,00: POKE 32765,76:
      POKE 32766,44: POKE 32767,254:
      REM CLEAR Y REGISTER BEFORE MOVE! [J]

220   POKE 32768,0: POKE 60,00: POKE 61,128: POKE 62,254:
      POKE 63,128: POKE 66,01: POKE 67,128: CALL 32763:
      REM  ZERO $8000-$80FF [J]

230   POKE 49159,0: POKE 49163,0:
      REM READ INTERNAL C3 AND CX ROM [J]
```

PROGRAM 10-1 cont

```
240  POKE 60,00: POKE 61,193: POKE 62,255: POKE 63,207:
     POKE 66,00: POKE 67,129: CALL 32763:
     REM  MOVE $C100-$CFFF [J]

250  POKE 49158,0: POKE 49162,0:
     REM READ USUAL C3 AND CX SLOTS [J]

260  PRINT : PRINT "[D]BSAVE IIEMON.C,A$8000,L$1000
270  REM : SAVE $C100-$CFFF TO DISK [J]

299  REM  [J]
        .................... [J]

300  REM : $D000-DFFF GRAB
310  POKE 60,00: POKE 61,208: POKE 62,255: POKE 63,223:
     POKE 66,00: POKE 67,128: CALL 32763:
     REM  MOVE $D000-$DFFF [J]

320  PRINT : PRINT "[D]BSAVE IIEMON.D,A$8000,L$1000
399  REM  [J]
        .................... [J]

400  REM : $E000-EFFF GRAB [J]
410  POKE 60,00: POKE 61,224: POKE 62,255: POKE 63,239:
     POKE 66,00: POKE 67,128: CALL 32763:
     REM MOVE $D000-$DFFF [J]

420  PRINT : PRINT "[D]BSAVE IIEMON.E,A$8000,L$1000
499  REM  [J]
        .................... [J]

500  REM : $F000-FFFF GRAB [J]

510  POKE 60,00: POKE 61,240: POKE 62,255: POKE 63,255:
     POKE 66,00: POKE 67,128: CALL 32763:
     REM  MOVE $F000-$FFFF [J]

520  PRINT : PRINT "[D]BSAVE IIEMON.F,A$8000,L$1000
599  REM  [J]
        .................... [J]

600  TEXT : HOME : CLEAR
610  FOR N = 1 TO 30:ZZ =  PEEK (49200) +  PEEK (49200) +
     PEEK (49200): NEXT N: REM BRACK [J]

900  PRINT "MONITOR SNATCH COMPLETE": PRINT : END
910  REM   [J]
         .................... [J]

2000  REM : TUTORIAL AND PROMPT [J]
```

PROGRAM 10-1 cont

```
2008  POKE 49167,0: REM  ALTSET ON
2010  VTAB 1: HTAB 12: FOR NN = 1 TO 15:
      PRINT  CHR$ (127);: NEXT NN: PRINT
2012  HTAB 12: PRINT  CHR$ (127);"  SNATCHMON  ";
      CHR$ (127)
2014  HTAB 12: FOR NN = 1 TO 15: PRINT  CHR$ (127);:
      NEXT NN: PRINT
2015  PRINT: PRINT
2016  PRINT "This program 'captures' the Apple IIe's monitor
      ROM into four 4Kx8 binary files named IIEROM.C
      thru IIEROM.F": PRINT
2017  POKE 1677,162: POKE 1686,162:
      REM  REAL QUOTES WITHOUT TEARS
2018  PRINT "Default use address is $8000, as needed by the
      MPC ap-ep EPROM burner.": PRINT : PRINT
2020  PRINT "Please insert SAVE disk into Drive 1.": PRINT :
      PRINT : PRINT "Then press  < space > to CONTINUE": PRINT
2022  HTAB 15: PRINT "-or-": PRINT
2024  HTAB 13: PRINT "< escape> to ABORT": PRINT : PRINT
2026  HTAB 13: PRINT "---< >---"; CHR$ (08); CHR$ (08);
      CHR$ (08); CHR$ (08); CHR$ (08);
2028  GET Z$
2030  IF Z$ >  < " " THEN  TEXT : HOME : CLEAR : END
2040  TEXT : HOME
2050  POKE 49166,0: REM  PRIMARY CHARACTER SET
2100  RETURN
```

Gotchas: [D] means "imbed an "escape-D" into your listing.
[J] means "imbed an "escape-J" into your listing.

This program is best run in the "old" 40-column mode with the 80-column firmware inactive.

the normal I/O. Soft switch $C00B turns ON the monitor for $C300 access, while $C00A turns ON the normal slot 3 usage.

Note that these switch flips are *backwards* from page 214 of the *Apple IIe Reference Manual*.

To repeat . . .

> Soft switch $C006 turns ON the usual $CX00 I/O space, and turns OFF the monitor "CD" ROM. (Decimal 49158)
>
> Soft switch $C007 turns OFF the usual $CX00 I/O space, and turns ON the monitor "CD" ROM. (Decimal 49159)
>
> Soft switch $C00A turns ON the usual $C300 I/O space, and turns OFF the monitor "CD" ROM. (Decimal 49162)
>
> Soft switch $C00B turns OFF the usual $C300 I/O space, and turns ON the monitor "CD" ROM (Decimal 49163)

Be sure you correct your reference manual. Very bad things happen if you get these actions mixed up. Note that you must *write* to these locations to activate them, such as with a "C006:00 [cr]" or a "POKE 49158,0 [cr]".

Reads just won't hack it.

There are a few other soft switches used to control the monitor ROM area, but we need not worry too much about these here. Normally, a cold boot is done before you try running SNATCHMON, which flips the switches into their "normal" positions. You then flip soft switches $C007 and $C00B long enough to move the $C100–CFFF monitor image to $8100–8FFF, and then flip these switches back to their usual $C006 and $C00A positions.

SNATCHMON assumes you have a single disk drive. A "press the spacebar" prompt lets you change diskettes if you want the images of IIEMON.C, IIEMON.D, IIEMON.E, and IIEMON.F to end up on a different diskette.

MODIFYING THE IIe MONITOR

The IIe monitor is crammed full and has very little in the way of free bytes. There are a few bizarre bugs, such as a programmer's ASCII first name that can get executed as op-codes by certain programs accessing <ctrl>-Y at the old location. The percentage of "compatible" II+ programs is far lower than Apple cares to admit, since just about any decent II+ program makes use of "illegal" monitor access. As a general rule, schlock II+ programs are IIe compatible, while creative and useful ones are not.

Areas available for rewriting are the cassette routines, the screen message, and the "Bryan" code. I urge you not to clobber the cassette routines, since cassettes can often salvage a disaster, even after all else has failed.

About the only monitor area that nobody, but nobody, wants to keep is the hole blaster from $C249 to C260.

So, as an example of patching the monitor, we will overwrite this hole blaster with code that accesses the old monitor.

The source code appears as Program 10-2, and is called KREBF SPELL.SOURCE, and assembles into the patch called KREBF SPELL. This module is simple enough that we can forego a flowchart.

This EDASM program provides the patch needed to overwrite the "hole blaster" in the monitor with a timed access to the "old" monitor entry point.

You get to location $C249 on a "[ctrl] <open-apple> [reset]. We first blast the warm start location at $03F4 by incrementing it. This tells the Apple that we are to do a

PROGRAM 10-2 Listing of KREBF SPELL.SOURCE.

```
----- NEXT OBJECT FILE NAME IS KREBF SPELL

8249:  8249            ORG    $8249     ;  Overwrites $C249 of IIE ROM CD

8249:             ;   *****************************************
8249:             ;   *                                       *
8249:             ;   *          -< KREBF SPELL >-            *
8249:             ;   *                                       *
8249:             ;   * ( REPAIR WILLFUL DAMAGE BY IIe ROM )  *
8249:             ;   *                                       *
8249:             ;   *      Version 1.0  ($C249-$C260)       *
8249:             ;   *                                       *
8249:             ;   *               1-24-84                 *
8249:             ;   *.......................................*
8249:             ;   *                                       *
8249:             ;   *          Copyright c 1984 by          *
8249:             ;   *                                       *
8249:             ;   *     Don Lancaster and SYNERGETICS     *
8249:             ;   *     Box 1300, Thatcher AZ., 85552     *
8249:             ;   *                                       *
8249:             ;   *            (602) 428-4073             *
8249:             ;   *                                       *
8249:             ;   *     All commercial rights reserved    *
8249:             ;   *                                       *
8249:             ;   *****************************************



8249:             ;           *** WHAT IT DOES ***

8249:             ;   This patch gives you an alternate to the stock
8249:             ;   IIe system monitor when properly burned into a
8249:             ;   2764 EPROM replacement for the CD monitor ROM.

8249:             ;   There are two improvements over the original:
8249:             ;
8249:             ;     1.  There is no hole blasting or other willful
8249:             ;         damage done on cold reboot.
8249:             ;
8249:             ;     2.  Holding the open-apple key down for four
8249:             ;         or more seconds on a cold reboot will drop
8249:             ;         you into the "old" monitor at the "old"
8249:             ;         entry point, for immediate "*" access.
8249:             ;
```

PROGRAM 10-2 cont

```
8249:     ;           *** HOW TO USE IT ***


8249:     ;   To INSTALL the KREBF spell:
8249:     ;
8249:     ;     1.  BLOAD an image of IIEMON.C into $8000-8FFFF.
8249:     ;         Then BLOAD KREBF SPELL  Then BSAVE KREBFMON.C,
8249:     ;         A$8000, L$1000.
8249:     ;
8249:     ;     2.  Use KREBFMON.C as a buffer image to burn the
8249:     ;         bottom half of a 2764 EEPROM.  Use IIEMON.D
8249:     ;         to burn the upper half.  Verify and then
8249:     ;         replace the IIe CD monitor with this 2764.

8249:     ;   To USE the KREBF spell:
8249:     ;
8249:     ;     1.  Do a cold reboot in the normal way for normal
8249:     ;         uses.  Just don't stay on the open-apple key
8249:     ;         for a ridiculously long time.
8249:     ;
8249:     ;     2.  To drop into the "old" monitor, do a cold
8249:     ;         reboot in the normal way, EXCEPT hold down
8249:     ;         the open-apple key until the "*" appears.


8249:     ;           *** GOTCHAS ***

8249:     ;   Entry into the old monitor is with the I/O space
8249:     ;   disabled and the CX00 ROM enabled.
8249:     ;
8249:     ;   To regain control of I/O, enter "C006:00<cr>",
8249:     ;   after dropping into the old monitor.
8249:     ;
8249:     ;   The 2764 used MUST be a 28 pin INTEL or HITACHI
8249:     ;   brand with an access time of 250 nanoseconds.
8249:     ;
8249:     ;   Do NOT use MOTOROLA or TEXAS INSTRUMENTS 2764's,
8249:     ;   since they are not and never were.
8249:     ;
8249:     ;   EPROM burning methods vary with the programming
8249:     ;   card design.  See Enhancement #10 in the SAMS
8249:     ;   Enhancing your Apple II series for more detail.



8249:     ;          *** ENHANCEMENTS ***

8249:     ;   Other, more powerful monitor modifications can
8249:     ;   be made using the same techniques shown here.
```

PROGRAM 10-2 cont

```
8249:              ;
8249:              ;    Similar ideas can be applied to the EF monitor
8249:              ;    ROM for a totally custom system.
8249:              ;
8249:              ;    By diverting the NMI to your own code, you can
8249:              ;    replace much of what a memory image card does
8249:              ;    at a tiny fraction of the cost.


8249:              ;              *** RANDOM COMMENTS ***


8249:              ;    The "KREBF" Spell for repairing willful damage
8249:              ;    originally appeared in Infocom's ENCHANTER.
8249:              ;
8249:              ;    It turns out enhancers need this spell even
8249:              ;    more than enchanters do.


8249:              ;                 *** HOOKS ***

8249:  007D        CSUMFIX    EQU    $7D       ; Must force $0BA8 checksum
8249:  FF59        OLDRST     EQU    $FF59     ; Old reset entry point
8249:  C061        OPENAPL    EQU    $C061     ; Open apple key
8249:  03F4        PWREDUP    EQU    $03F4     ; Powerup funny EOR
8249:  FCA8        WAIT       EQU    $FCA8     ; Monitor time delay


8249:              ;            *** KREBF SPELL ***
8249:              ;
8249:              ;    The KREBF SPELL is a patch that overwrites
8249:              ;    the hole blaster in the original "CD" ROM.
8249:              ;
8249:              ;    The code is replaced with a timer loop that
8249:              ;    causes a jump to the "old" monitor routine
8249:              ;    if the open-apple key is held down for more
8249:              ;    than four seconds after a cold reboot.
8249:              ;
8249:              ;    Actual time delay is done by the monitor
8249:              ;    WAIT subroutine.
8249:              ;
8249:              ;    Note that the checksum correction byte must
8249:              ;    force an overall $0BA8 checksum for all bytes
8249:              ;    used in this patch.
```

PROGRAM 10-2 cont

```
8249:EE F4 03          START   INC  PWREDUP     ; Abort warm restart

824C:A0 1C                     LDY  #$28        ; For 40/10ths of a second

824E:A9 C5             RETRY   LDA  #$C5        ;  for one tenth of a second
8250:20 A8 FC                  JSR  WAIT        ;  stall via monitor delay
8253:2C 61 C0                  BIT  OPENAPL     ;  read the open-apple key
8256:10 0C        8264         BPL  START+$1B   ;  and exit on key release
8258:88                        DEY              ;  one less count
8259:D0 F3        824E         BNE  RETRY       ;  and keep it up
825B:4C 59 FF                  JMP  OLDRST      ;  go old monitor on timeout

825E:7D                        DFB  CSUMFIX     ; Fix diagnostics checksum

825F:00 00                     DFB  $00,$00     ; Two bytes complete fill
```

cold reboot, rather than letting the applications program in use "capture" the reset back to itself.

Then, we run through a timing loop 40 times. Each time through, we stall for one-tenth of a second via the monitor WAIT routine at $FCA8. We then check to see if the open-apple key is released. If it is released, we continue with a stock cold reboot, sans the hole blasting. If the key is down, we keep on stalling.

If we timeout to a full four seconds, we instead jump directly to the "old" reset code which, thankfully, remains in the monitor at $FF59. The four seconds was selected to be long enough that a child or an unknowing user is unlikely to accidentally trip the old monitor access.

There are some undocumented Apple IIe diagnostics stashed at $C400–CFFF that will fail the "CD" ROM if a checksum of all the bits in the entire ROM does not agree with a magic value.

To pass these diagnostics, the checksum of the KREBF SPELL must be the same as that of the hole blaster bytes it replaced. The magic checksum for these $18 bytes of code is $0BA8. The byte at $825E corrects this checksum to let your altered code pass the diagnostics.

Note that any changes at all to this code will need a different value for the checksum adjustment byte.

To use your source code, you assemble it into the patch, and then overwrite the patch onto your monitor image. Alternately, you can hand-load the patch over the monitor code. Details on this appear in Chart 10-1.

Basically, what you do is BLOAD IIEMON.C. Then you BLOAD KREBF SPELL. This patch overwrites the hole blaster with the "old" monitor access timer. Then you BSAVE KREBFMON.C, A$8000, L$1000. All of which casts the KREBF spell on your monitor image.

Chart 10-1. How to Install Your "Old" Monitor EPROM in an Apple IIe

1. Using the program SNATCHMON, place copies of IIe ROM images onto a work diskette under the filenames of IIEMON.C, IIEMON.D, IIEMON.E, and IIEMON.F.

2. Place a copy of KREBF SPELL onto the same work disk. Do this by copying the companion diskette, assembling from KREBF SPELL.SOURCE, or by simple hand loading.

3. Cast the KREBF spell this way:

   ] BLOAD IIEMON.C. <cr>
   ] BLOAD KREBF SPELL <cr>
   ] BSAVE KREBFMON.C, A$8000,L$1000 <cr>

4. If it is needed, plug the 2764 adapter into your EPROM burner. Be sure the notch points to the handle and that the grabber contacts the correct point in the burner. In the MPC ep-ap, the grabber goes to pin 1 of U10.

   WARNING: BE SURE YOUR BURNER IS CONFIGURED FOR AN INTEL 2764 BURN (no adapter) OR A 2732 BURN (with adapter) BEFORE CONTINUING.

   The steps that follow assume you are using the adapter on an older burner, whose 2732 work buffer goes from $8000–8FFF.

5. Erase a 28-pin, 250 nanosecond INTEL or HITACHI 2764 EEPROM using a suitable ultraviolet source.

6. Insert the 2764 EPROM into the adapter socket. Be extra careful not to bend any pins. New 2764s will need all their pins "rocked" inwards to fit.

7. Flip the adapter switch to "0" or towards pin 14. Then arm the EPROM burner and boot the support code.

8. Verify the erasure. Then, load, burn, and verify KREBFMON.C as if it were intended for a 2732 EPROM.

9. Flip the adapter switch to "1" or towards pin 28. Then load, burn, and verify IIEMON.D as if it were intended for a 2732 EPROM. Note that the Krebf spell does NOT get cast on the "D" side.

10. Turn the Apple supply power off and remove BOTH ends of the line cord. Carefully remove the CD ROM at E8, using an IC puller, while keeping one wrist on the top of the Apple power supply. Store this ROM in protective foam for possible later warranty repairs.

11. Remove the newly programmed 2764 from the burner and insert it in the newly emptied socket on your IIe, being very careful that the notch goes TOWARD the keyboard and no pins are bent or tucked under. Keep your wrist on the power supply as you do this.

12. Apply supply power and run the following checks:

    (a) Normal cold boot of NONVALUABLE diskette in drive 1 when power is applied.

    (b) Normal reset to Applesoft or applications program on <ctrl>-<reset>.

Chart 10-1—cont. How to Install Your "Old" Monitor EPROM in an Apple IIe

(c) Normal cold reboot on <ctrl>-<open-apple>-<reset>.

(d) Drop into the monitor on <ctrl>-<open-apple>-<reset> with the <open-apple> key held down for four seconds **after release of the reset key.** Wait for the beep and the "*" symbol.

(e) From the monitor, do a C006:00 <cr>, followed by a 6-<ctrl>-P. You should get a standard cold reboot.

BE SURE TO USE THIS "C006:00 <cr>" AT THE START OF EACH MONITOR ACCESS. OTHERWISE, THE I/O SPACE WILL NOT GET PROPERLY ACTIVATED.

(f) Do a <ctrl>-<closed-apple>-<reset>. After some screen flashing, you should get the "KERNEL OK" message.

13. Put some opaque tape over the lid of your new "old" monitor. Add a bright dot to remind yourself of this change. Save the CD ROM should warranty repairs ever be needed. This completes your installation.

You then program your 2764, using KREBFMON.C for the low half (switch left) and IIEMON.D. for the upper half (switch right). For this particular use, you need only replace the CD ROM with a 2764 EPROM. The "EF" ROM does not get changed.

Note that the KREBF spell is cast *only* on IIEMON.C, and *not* on the other monitor images.

The Applesoft program SNATCHMON, source code KREBF SPELL.SOURCE, and the machine language binary patch KREBF SPELL are included in the companion diskette to this volume.

Since I cannot legally sell you copies of the IIe monitor image if you do not personally own a IIe on which those images are to be uniquely used, these images are not available from me. You will, instead, have to grab your own with SNATCHMON. The painless and automatic process takes less than a minute.

Selling ready-to-go 2764 modified EPROMs would also be considered a no-no in some quarters, despite its instant cash potential. I'll pass on this as well.

It is very interesting to compare Programs 10-1 and 10-2. Assembly language Program 10-2 was written and debugged much faster than was Applesoft Program 10-1. Program 10-2 does more. And its documentation is far more legible.

Which, of course, proves once again that assembly language is not only better than Applesoft, but cleaner and faster to program as well, besides being vastly easier to understand and far cleaner to document.

If you are weak on assembly language programming, check into *Don Lancaster's Assembly Cookbook for the Apple II/IIe* (Sams Cat. No. 22331) for some solid, thorough, and easy-to-understand text and ready-to-use support modules.

TESTING AND EXTENDING

Step 12 of Chart 10-1 gives you a detailed checkout routine. Be very careful when you swap EPROM for ROM that you bend no pins and that the code dot and notch goes towards the keyboard, or front, of your IIe. Save the old "CD" ROM in protective foam should repairs be needed.

Be sure to turn power off, and remove BOTH ends of the Apple IIe line cord. Always lean on the top of the power supply with your wrist before inserting or removing any integrated circuit. Just to be darn sure, hold the Apple end of the line cord in your hand as you do this.

It might be best to program your EPROMs on one Apple and test them on a second IIe. This separates programming bugs from operational problems.

Your checkout procedure should include a cold boot, a cold reboot, a cold reboot into the monitor, a cold reboot from the monitor, and a diagnostics pass using the closed-apple key.

One very important gotcha . . .

> When you drop into the "old" monitor, you do so with the I/O space disabled and the "CD" ROM enabled.
>
> ALWAYS do a C006:00 <cr> as your very first "old" monitor instruction!

This sounds slightly flakey, but I haven't found a really good way around it. Maybe you can. At any rate, this is no worse than good old CALL -151; in fact it is one keystroke shorter. Besides, it *always* works.

Any time, any place, any program, any reason.

If you do not activate the I/O space, all of the usual monitor functions will work in the usual way. The only little problems are that you will have no DOS, printer, or other I/O slot access available. So, be sure to get in the habit of whapping $C006 on each old monitor entry.

There are lots of possible things you can do with a custom monitor. For instance, you can rearrange Applesoft to suit yourself. You can also do much of what a "snapshot" card can for a tiny fraction of the price. To do this, grab the NMI vector and dump everything in sight to the stack, including all registers, all soft switches, hard to read locations such as $0200, and some identifying markers. Then exit to some code that uses none of the usual page zero or screen scrolling locations.

You can also replace the entire operating system with your own custom code. This could prove most useful for dedicated data acquisition, certain CAD/CAM programs, and "turnkey" applications in general.

What other monitor mods would you like to see?

As a reminder, separate and different patches are needed for a IIc or a "new" IIe absolute reset. Contact me directly via the helpline for further info on this.

> The following programs appear on the support diskette for this volume:
>
> SNATCHMON
> KREBF SPELL.SOURCE
> KREBF SPELL
>
> See the response cards in the back of the book for full details.

This enhancement works on all Apples. It is useful by both cheaters and survivors.

Enhancement 11

CASTLE WOLFENSTEIN® ESCAPE MAPS

Having trouble getting past the lowly rank of buck private? SS troubling you? This complete set of maps and playing hints might be just what you need.

CASTLE WOLFENSTEIN ESCAPE MAPS

How to Create Your Own Escape Maps

1. Two sets of maps are provided. The set with the white borders stays in this book and should not be removed.
2. Carefully cut out ONLY the three pages that have grey borders. Cut first on the dashed line and then carefully trim away all outside gray.
3. Get six sheets of laminating plastic from an office supply or variety store. Separately laminate each of the three cut sheets. Leave a wide border on all four edges. Round all corners.
4. To use your maps, mark on them with a grease pencil, an erasable blackboard felt marker, or an overhead projector crayon. Use suitable symbols for war plans, bullets, uniforms, grenades, SS, bulletproof vests, knockwurst, etc.

Sams

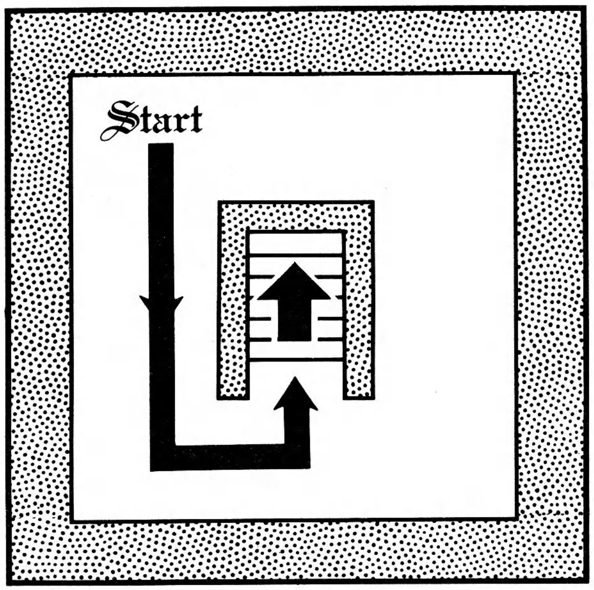
Castle Wolfenstein escape route

level I

At the very start, nothing happens until you move or aim your gun. So, TAKE YOUR TIME! Wait until the guards are in good positions before you make your first move.

The only way out of this level is up the stairs. If things look hopeless, give up and try again. Your position in the first room is randomized on each replay.

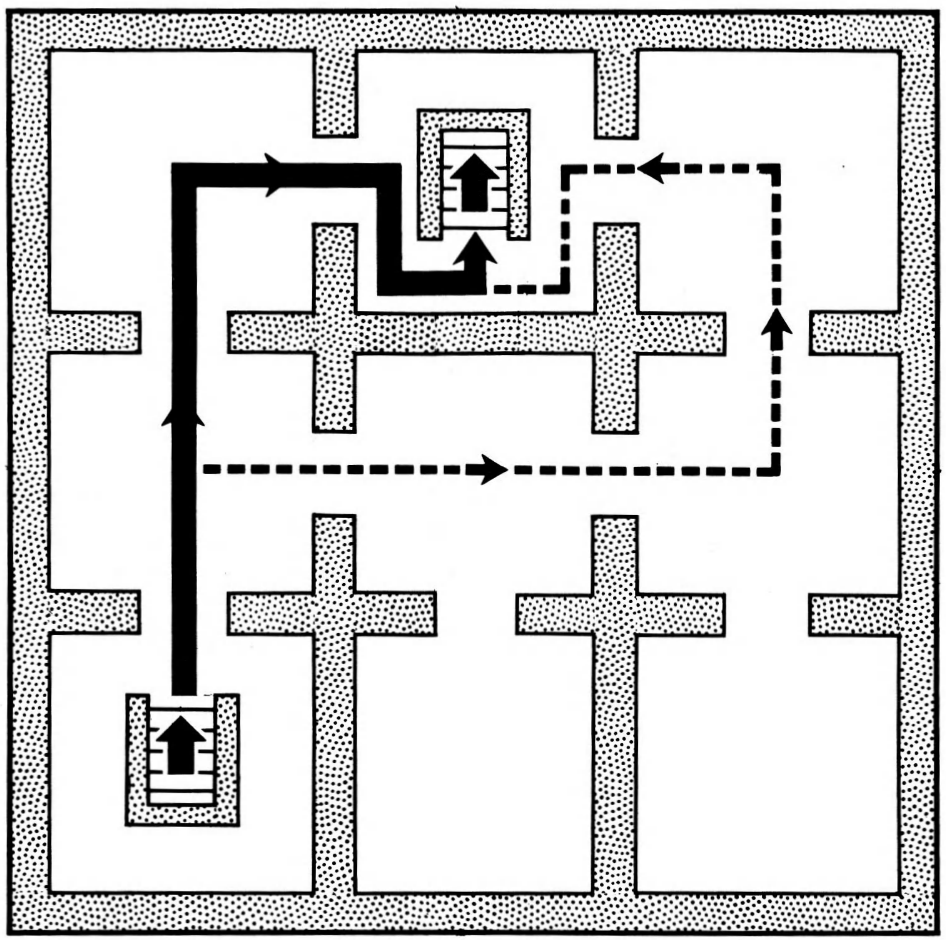
Try not to kill the guards in the first room. Wait until they are out of your path and then run for the stairs. You can later return in uniform and check any chests.



If you are locked in a closet, fire your gun once at the lock. If the door does not open, wait for a guard to open it for you.

Sams

Sams



Castle Wolfenstein escape route

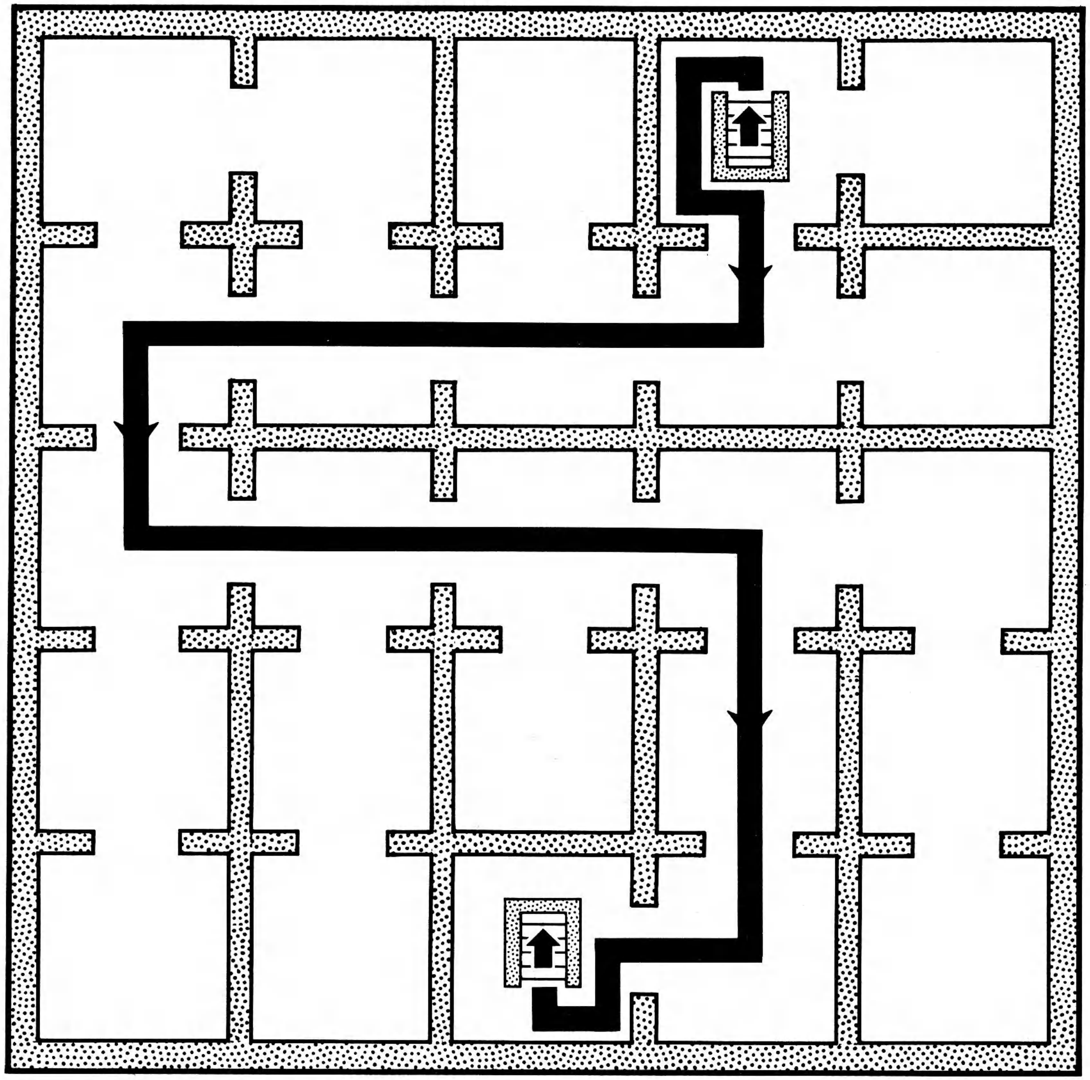
level II

There are two routes through this level. Try the longer route if there are avoidable SS on the main route.

Chest opening time can be sped up by leaning on the IIe spacebar or by using the II+ repeat key.

Try not to go beyond this level unless you are wearing a uniform and a bulletproof vest.

Sams



Castle Wolfenstein escape route

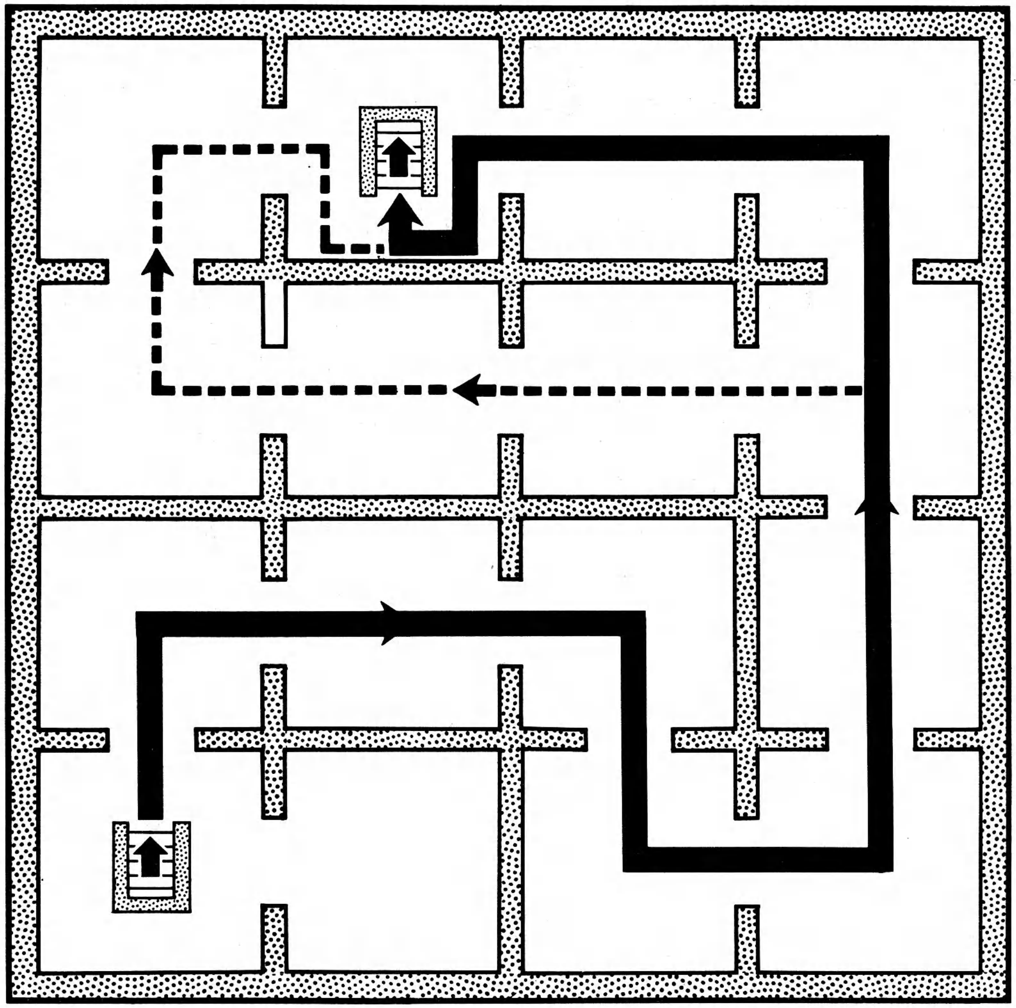
level II

There is only one reasonable way through this level, but that route is twelve rooms long.

Avoid side trips on your first pass through, unless you are in desperate need of something. If you must, you can return later.

Guards normally will not bother you if you are wearing a uniform and a bulletproof vest, and if you have not drawn your gun. Neither guards nor SS can normally see through walls or partitions.

Sams



Castle Wolfenstein escape route

level IV

There are two routes through this level. Try the longer route if there are avoidable SS on the main route.

Grenades are best used to provide "shortcuts" through interior walls. You will need one grenade thrown vertically, or two thrown horizontally.

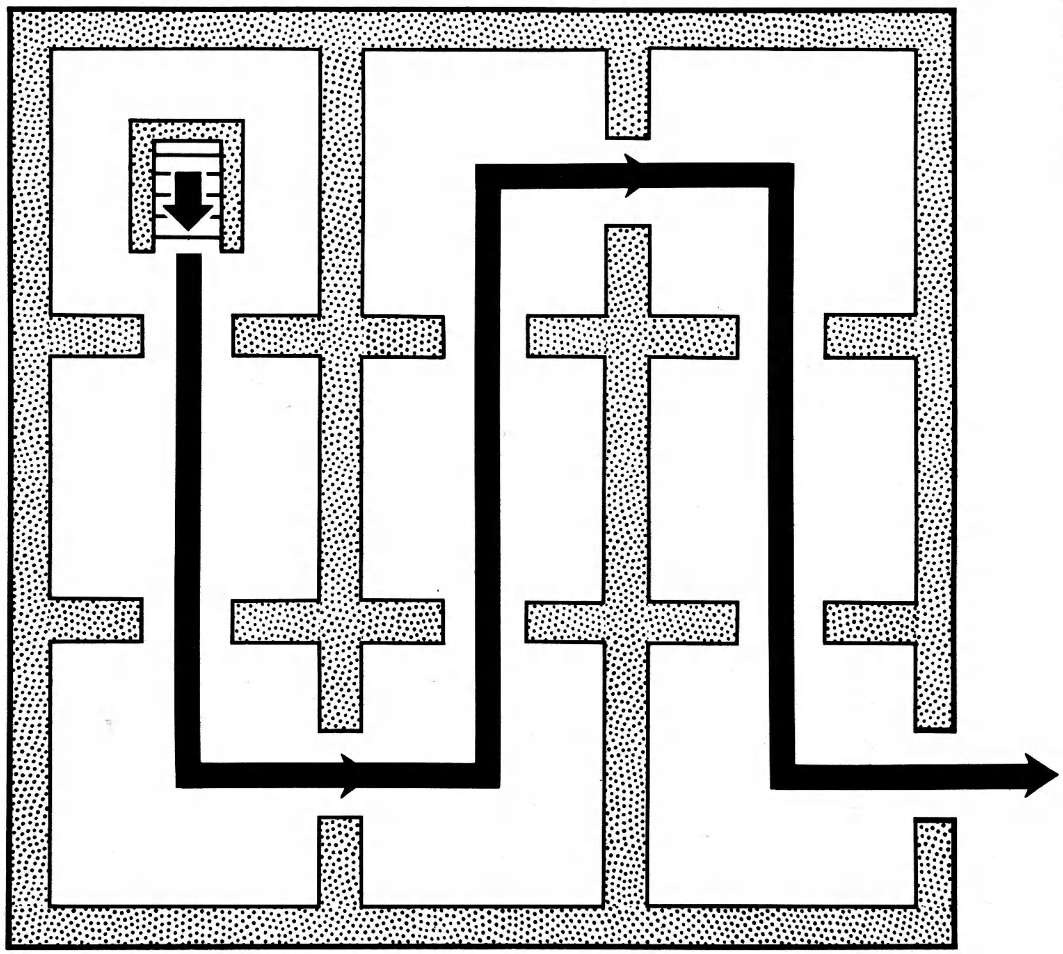
You can hold up a guard without killing him to increase your stock of bullets or grenades. Try this only when SS are not present and when you can easily exit the room.

Sams

Sams

Castle Wolfenstein escape route

V level



This level requires very little thought. All you have to do is "run the gauntlet" of all nine rooms in order.

Do not exit the last room without the war plans, or you will forever lose the present castle.

Escape!

Be sure to TURN POWER OFF after your last session. In the real world, Wolfenstein SS have brutally murdered many Visicalc files!

Sams

Castle Wolfenstein playing hints

The playing disk is VERY fragile and VERY easy to destroy. Use only your second or third backup copy. Backups can be made by any of the usual methods.

AVOID WANTON KILLING! The faster you travel and the fewer troops you blow away, the better your odds of survival.

Always BRIEFLY enter a new room and then leave IMMEDIATELY. This gives you a quick glance at what to expect, without arousing too much attention.

The outside walls of the castle do not change with a "new" castle layout. Only the room contents and inside partitions change. As you go up in rank, the number and speed of the SS will increase.

The guards have short memories. If you raise havoc in a room, leave and then return. Regular guards should go back to routine patrol when you do this.

Guards that seem to be blocking your way can be enticed to a new position by firing your gun once into the air. When the guard gets to where you want him, leave the room to freeze his new routine.

If you are beside a gernade when it goes off, you die. If you are within two steps, you lose your uniform, vest, and plans. Three or more steps away is safe.

Grenades can be used in pairs. Use the first one to blow a small hole in an inside wall. The second one can then be thrown through the new hole.

If you can get the drop on an SS, you can steal his vest. If you then leave the room and return, he gets demoted to an ordinary guard on routine patrol. Vest stealing only works on an SS who has not yet decided to chase you.

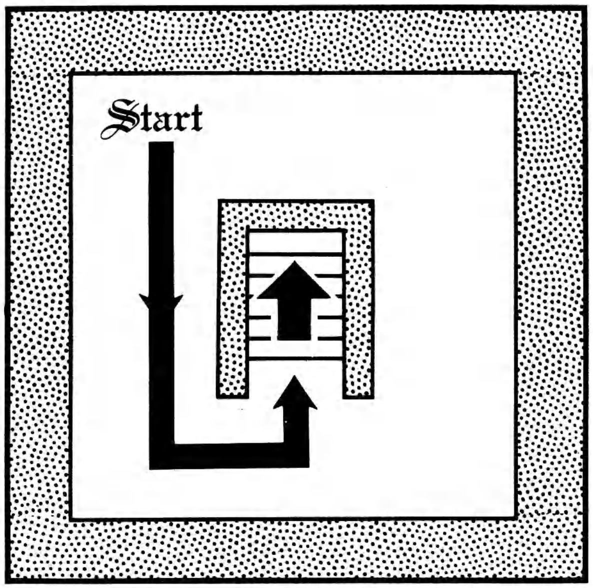
If the SS are chasing you, stop immediately at the entrance of the next room and plan ahead. SS will not enter a room unless you are at least three steps from the entrance.

A locked door can sometimes be opened by standing at the room entrance and firing your gun into the air. A guard will open the door for you as he investigates. Leave and then re-enter the room.

Neither guards nor SS will step over dead bodies.

Cut along dotted line. Then cut away all grey.

Sams



If you are locked in a closet, fire your gun once at the lock. If the door does not open, wait for a guard to open it for you.

Castle Wolfenstein escape route

level I

At the very start, nothing happens until you move or aim your gun. So, TAKE YOUR TIME! Wait until the guards are in good positions before you make your first move.

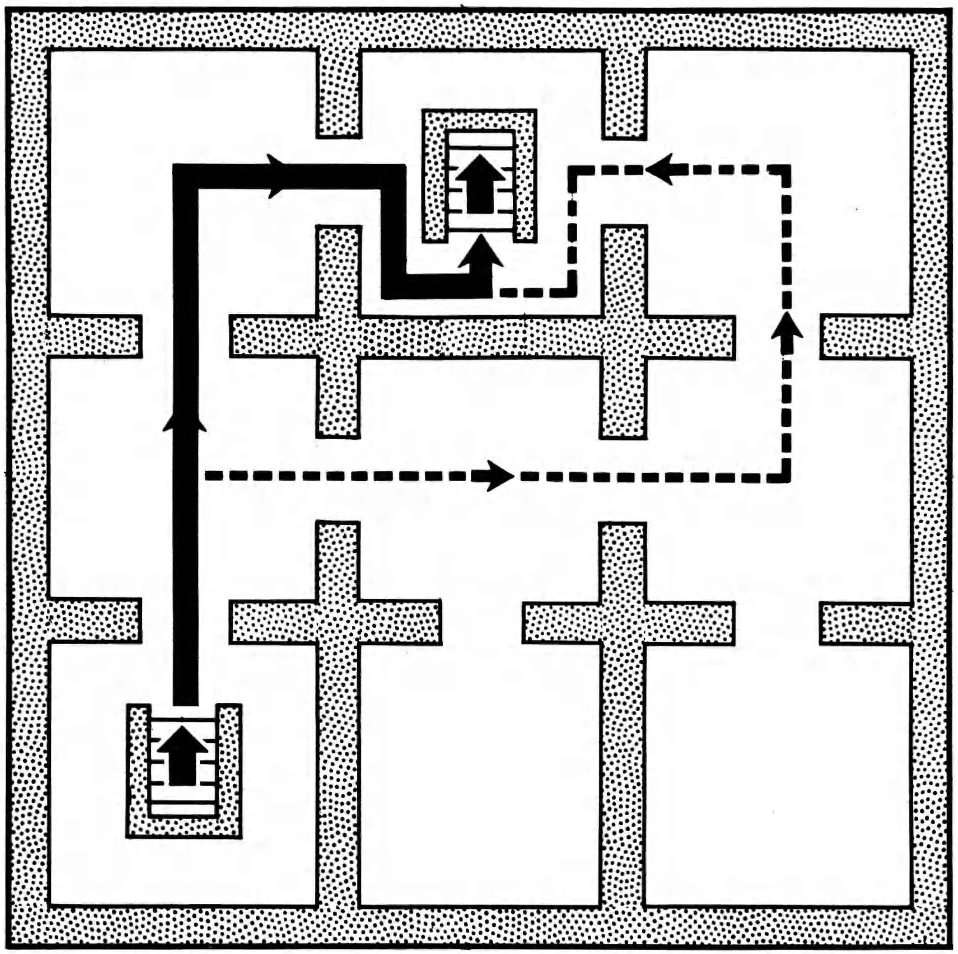
The only way out of this level is up the stairs. If things look hopeless, give up and try again. Your position in the first room is randomized on each replay.

Try not to kill the guards in the first room. Wait until they are out of your path and then run for the stairs. You can later return in uniform and check any chests.

Sams

Cut along dotted line. Then cut away all grey.

Sams



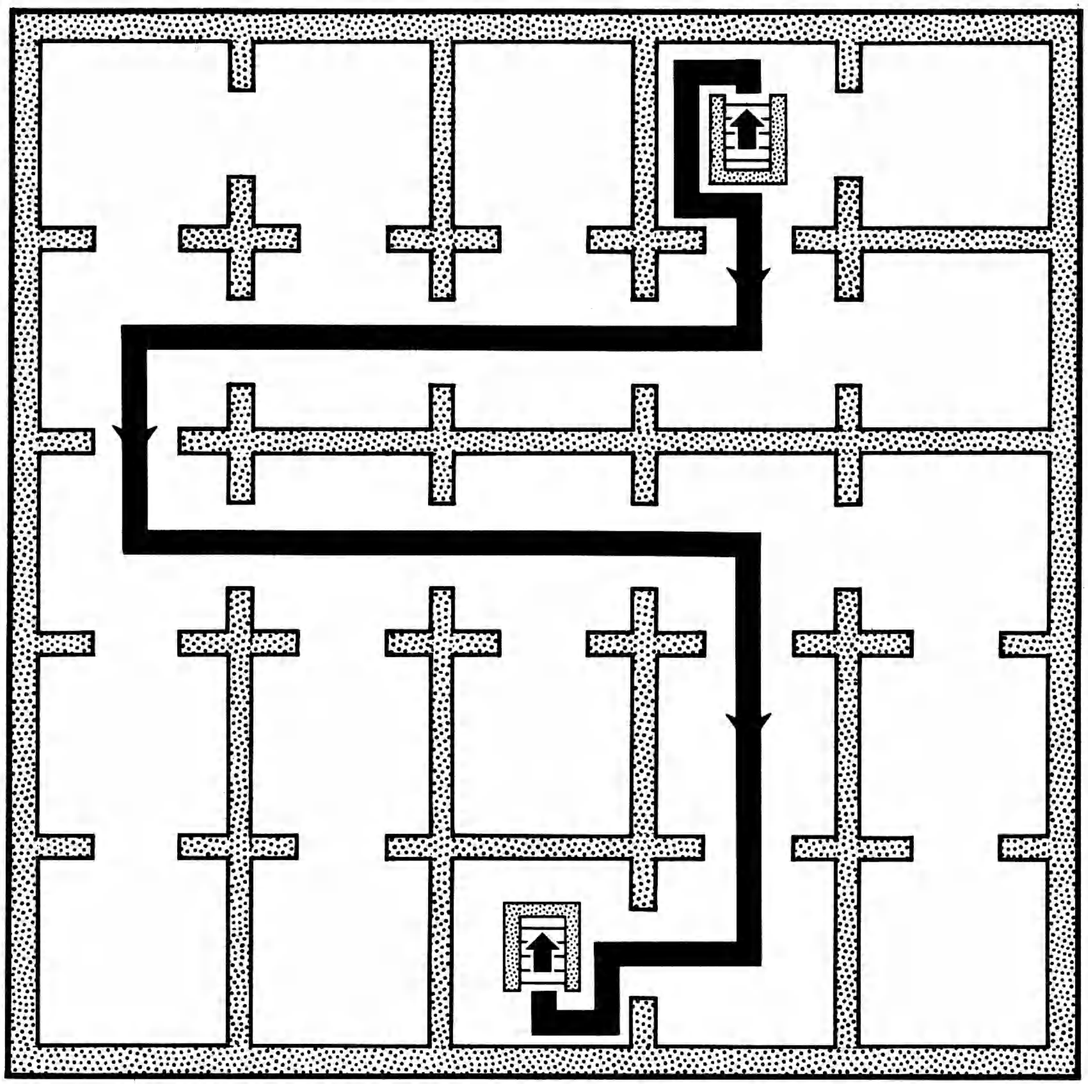
Castle Wolfenstein escape route

There are two routes through this level. Try the longer route if there are avoidable SS on the main route.

Chest opening time can be sped up by leaning on the IIe spacebar or by using the II+ repeat key.

Try not to go beyond this level unless you are wearing a uniform and a bulletproof vest.

Sams



Castle Wolfenstein escape route

III level

There is only one reasonable way through this level, but that route is twelve rooms long.

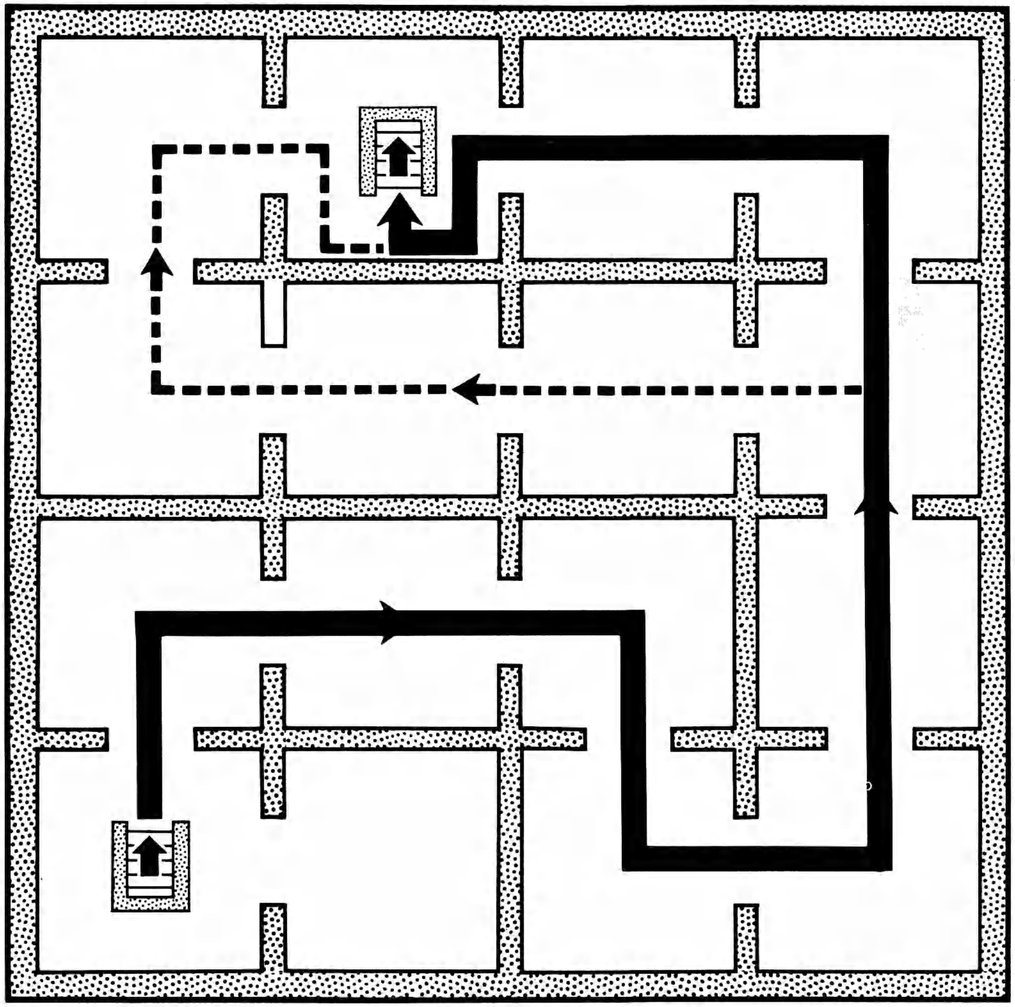
Avoid side trips on your first pass through, unless you are in desperate need of something. If you must, you can return later.

Guards normally will not bother you if you are wearing a uniform and a bulletproof vest, and if you have not drawn your gun. Neither guards nor SS can normally see through walls or partitions.

Sams

Cut along dotted line. Then cut away all grey.

Sams



Castle Wolfenstein escape route

level IV

There are two routes through this level. Try the longer route if there are avoidable SS on the main route.

Grenades are best used to provide "shortcuts" through interior walls. You will need one grenade thrown vertically, or two thrown horizontally.

You can hold up a guard without killing him to increase your stock of bullets or grenades. Try this only when SS are not present and when you can easily exit the room.

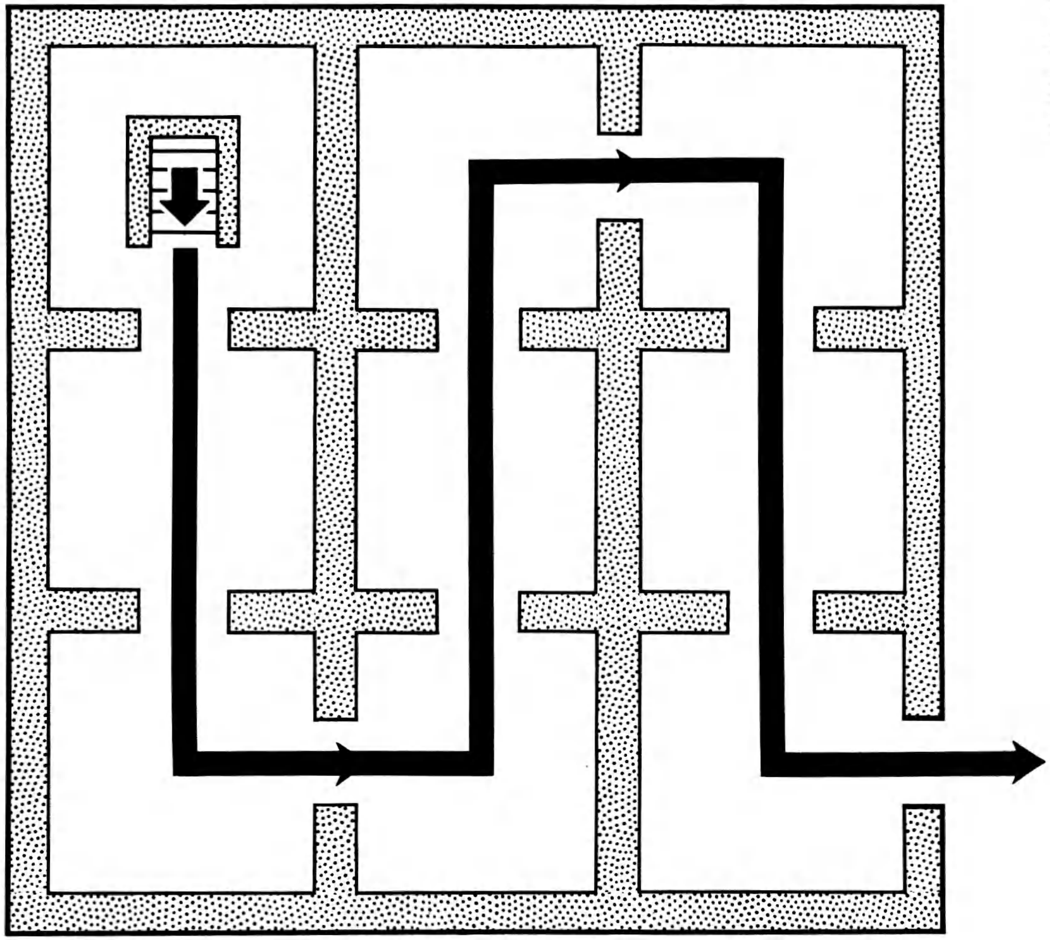
Sams

Cut along dotted line. Then cut away all grey.

Sams

Castle Wolfenstein escape route

level V



This level requires very little thought. All you have to do is "run the gauntlet" of all nine rooms in order.

Do not exit the last room without the war plans, or you will forever lose the present castle.

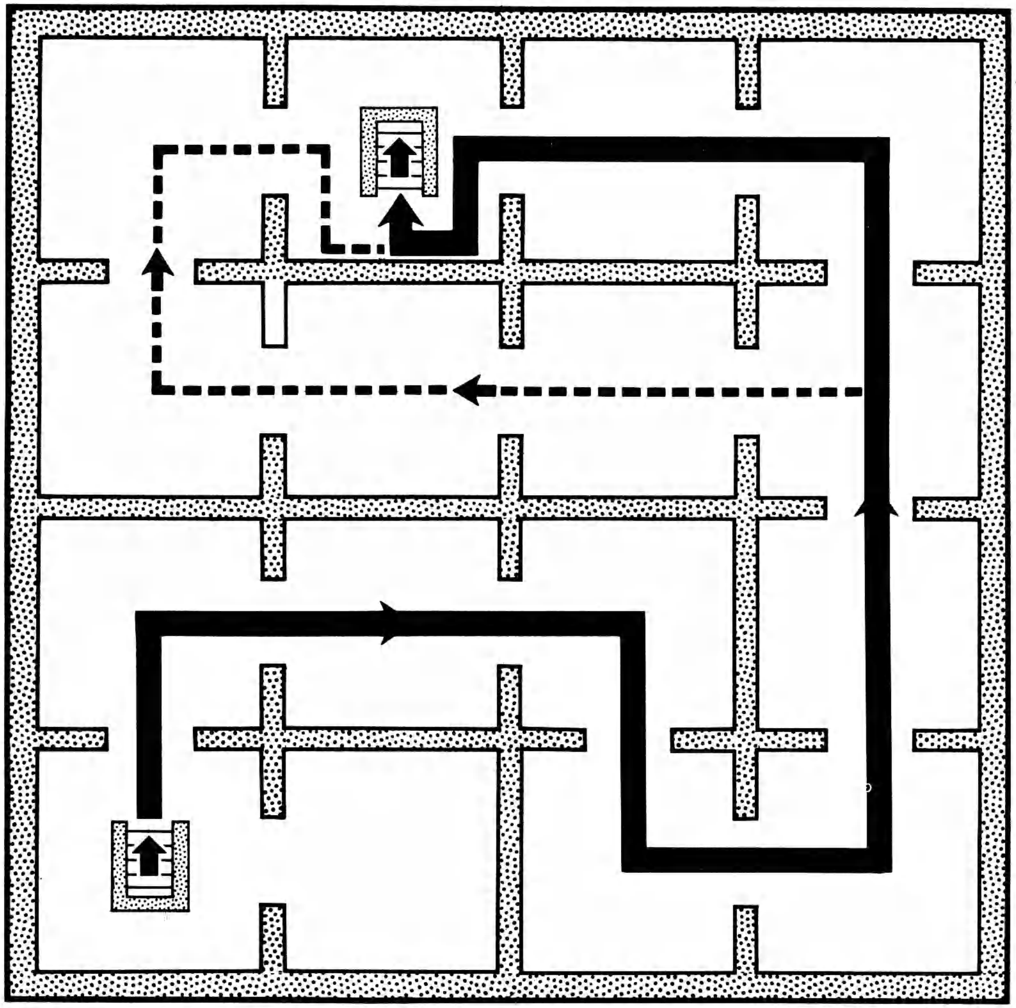
Escape!

Be sure to TURN POWER OFF after your last session. In the real world, Wolfenstein SS have brutally murdered many Visicalc files!

Sams

Cut along dotted line. Then cut away all grey.

Sams



Castle Wolfenstein escape route

IV level

There are two routes through this level. Try the longer route if there are avoidable SS on the main route.

Grenades are best used to provide "shortcuts" through interior walls. You will need one grenade thrown vertically, or two thrown horizontally.

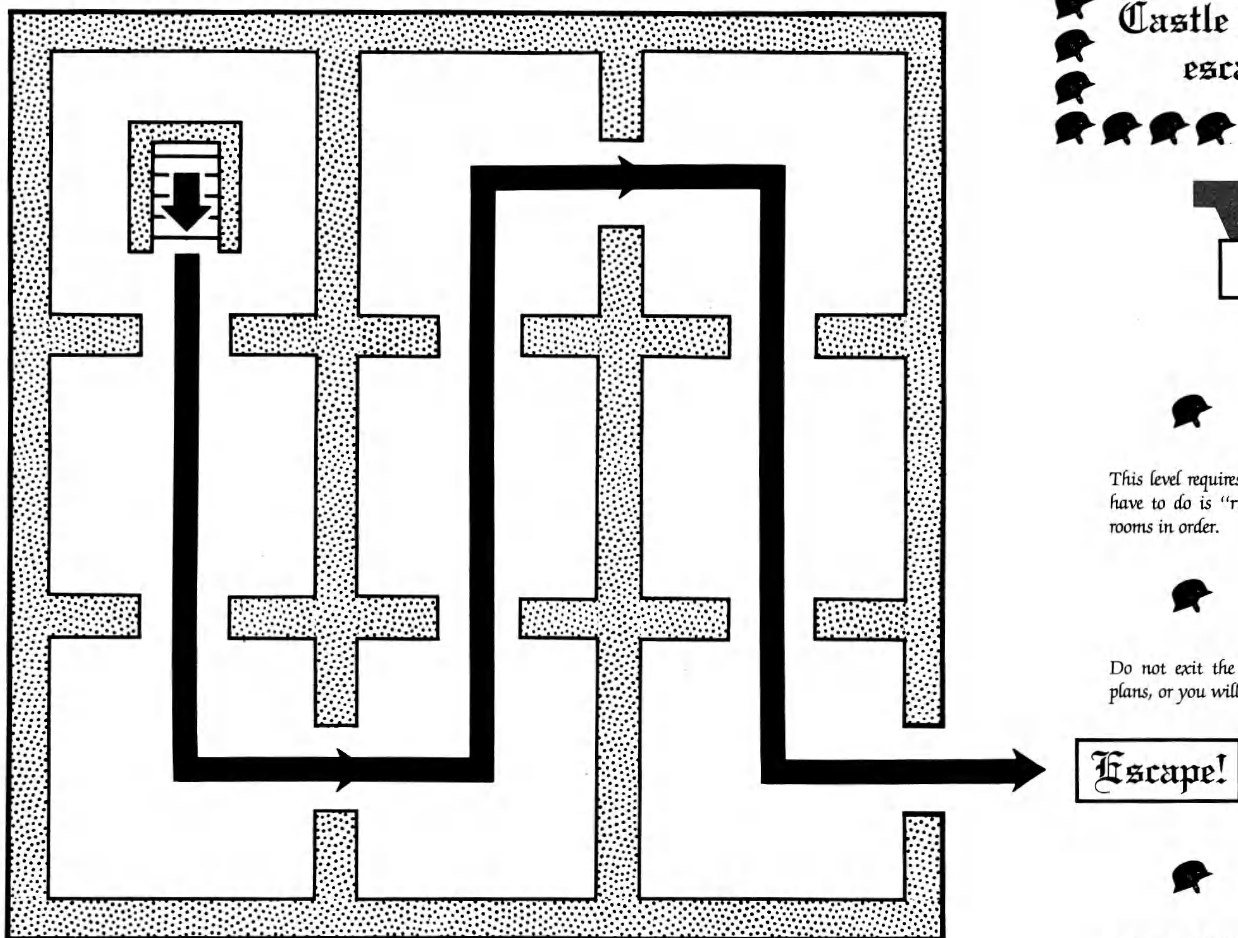
You can hold up a guard without killing him to increase your stock of bullets or grenades. Try this only when SS are not present and when you can easily exit the room.

Sams

Cut along dotted line. Then cut away all grey.

Sams

Castle Wolfenstein escape route



level V

This level requires very little thought. All you have to do is "run the gauntlet" of all nine rooms in order.

Do not exit the last room without the war plans, or you will forever lose the present castle.

Be sure to TURN POWER OFF after your last session. In the real world, Wolfenstein SS have brutally murdered many Visicalc files!

Sams

Cut along dotted line. Then cut away all grey.

Sams

Castle Wolfenstein playing hints

The playing disk is VERY fragile and VERY easy to destroy. Use only your second or third backup copy. Backups can be made by any of the usual methods.

AVOID WANTON KILLING! The faster you travel and the fewer troops you blow away, the better your odds of survival.

Always BRIEFLY enter a new room and then leave IMMEDIATELY. This gives you a quick glance at what to expect, without arousing too much attention.

The outside walls of the castle do not change with a "new" castle layout. Only the room contents and inside partitions change. As you go up in rank, the number and speed of the SS will increase.

The guards have short memories. If you raise havoc in a room, leave and then return. Regular guards should go back to routine patrol when you do this.

Guards that seem to be blocking your way can be enticed to a new position by firing your gun once into the air. When the guard gets to where you want him, leave the room to freeze his new routine.

If you are beside a gernade when it goes off, you die. If you are within two steps, you lose your uniform, vest, and plans. Three or more steps away is safe.

Grenades can be used in pairs. Use the first one to blow a small hole in an inside wall. The second one can then be thrown through the new hole.

If you can get the drop on an SS, you can steal his vest. If you then leave the room and return, he gets demoted to an ordinary guard on routine patrol. Vest stealing only works on an SS who has not yet decided to chase you.

If the SS are chasing you, stop immediately at the entrance of the next room and plan ahead. SS will not enter a room unless you are at least three steps from the entrance.

A locked door can sometimes be opened by standing at the room entrance and firing your gun into the air. A guard will open the door for you as he investigates. Leave and then re-enter the room.

Neither guards nor SS will step over dead bodies.

Sams

Cut along dotted line. Then cut away all grey.

This enhancement needs the 128K Apple IIe, but you can use the ideas here to crack any machine language code on any Apple.

Enhancement 12

TEARING INTO APPLE WRITER IIe

The "tearing method" of Enhancement 3 is used to give a very detailed and complete analysis of the single most significant Apple program of all time. You can use the results to create your own custom word processor that does exactly what you want.

TEARING INTO APPLE WRITER IIe

There have been lots of helpline requests to apply the "tearing method" of Enhancement 3 to a really heavy program. As you should know by now, the "tearing method" is an astonishingly fast way of breaking down, color coding, and then analyzing *any* machine language computer program running on *any* Apple.

Ready? Here we go . . .

The single most significant Apple IIe program of all time is *Apple Writer* IIe. This numero uno program far outsells all other Apple word processors. AWIIe even outsells *VisiCalc*® by a ridiculous margin.

The reasons are simple. Apple Writer IIe is the first "no excuses, no apologies" word processor available for the IIe, and is one of very few word processing packages that is genuinely fun to use and very easy to learn. It is also the *only* programmable word processor that I know of, which opens a mind-boggling array of largely unexplored new applications.

It did take Apple Computer three tries to finally do it right. But the third time seems to be a charm.

Very forthrightly, Apple elected to make Apple Writer IIe more or less of an "open" program. It is easily inspected and easily modified to suit your needs. Only the bare minimum of "protection" essential to preventing blatant ripoffs is built into the booting process.

Fortunately, after booting a legal copy of Apple Writer IIe, there is a fairly simple way to customize and modify the machine-resident program for your own special needs. This also gives you a method to market your own commercial patches in ways that are legal, professional, and practical.

Before you change anything, of course, you have to analyze it and see how it works.

Why would you want to tear into a major Apple program? Well, first and foremost, because it is there. It is absolutely inexcusable to *ever* use *any* computer program without doing a complete analysis of that program, and then capturing your own source code for study and custom modification. To not do this is utterly unthinkable.

Dumb, even.

Secondly, you just might be interested in seeing how things are done. Things such as managing both 64K banks of IIe memory, ringing the cute ding-dong, moving DOS to high RAM, understanding the WPL programmable interpreter, or just seeing how a master programmer handles things.

Thirdly, you may want to get rid of the warts. Like the "short line" problem. Or the "grungy underline" problem. Or the "old Epson" problem. Or the "HIRES dump" problem.

Lastly, you might want to customize Apple Writer IIe for your own needs. You might want to use a faster version of DOS. Or you might want to speed up the boot process so it exactly loads your machine without wasting time on configuration tests. Or maybe you would like to minimize disk swapping on a single drive system.

Perhaps you'd like to add the sorely needed space-on-disk routine. Or do a HIRES preview of nine pages at a time. Or add a printer buffer. Or you might just want to know what space is left over where in your machine for custom support programs. Or want to integrate spreadsheet or report code. You might want to know how to support an oddball 80-column card, an unusual printer interface, or a non-Corvus hard disk.

My own reasons for tearing into Apple Writer IIe are to push the limits. This is a great little word processor, but, would you believe it processes *pictures* even better than it does words? But, that's another story for another time. One that has a great plot.

Many, in fact.

Apple Writer IIe also interfaces beautifully with newly overhauled EDASM. This converts an impressive upgrade of what was a rather dreary and dumpy, old assembler into one of the finest macro assemblers available anywhere. Full details on "new way" editing appear in *Don Lancaster's Assembly Cookbook for the* Apple II/IIe (Sams Cat. No. 22331).

To push these limits, I want to modify WPL. I would like a single key WPL user input. I need the ability to "put the cursed character into the D$ string." More variables would be handy. So would be the ability to PEEK and POKE what you want where you want, and to link machine language modules. Some older printer interface cards demand this POKE capability. Tabbing in the EDASM "new way" editing could be dramatically sped up with a link to a custom machine language module.

Or, to get really wild, I want WPL to directly control a plotter whose pen has been replaced with a focusing photocell for fully automated picture-to-data conversions.

Font libraries anyone?

We will limit our analysis here to the "F" version of Apple Writer IIe, intended for use on a 128K Apple IIe with an extended 64K memory card in slot 0, running under slightly modified DOS 3.3e. The "F" version uses all 128K, while the "E" version uses only the main 64K. The booting process tests your machine and then picks which one you get.

If you are going to interface anything else to your word processor, the "F" version is the best choice, since it has a reasonable amount of uncommitted RAM left over. Should you not be using the "F" version, you can use the ideas here to analyze any word processor or any other machine language program on any system you choose.

THE TEARING METHOD REVISITED

Several reviewers have complained that the tearing method of Enhancement 3 takes some time and effort to master.

Gee.

Yes, your first two or three tearing ventures will take some time and effort. But once you have the method mastered, you'll find the tearing method to be at least 10 times faster than any other reasonable scheme for analyzing machine language programs. Picking the right program for your first tearing can make a big difference. For most people, either FID or BUGBYTER would be good choices.

Here are two key hints to improve your tearing attacks:

First, it often pays to work *backwards* from the end back to the beginning of a module. Loops leap out at you this way, and the good stuff a module does often happens fairly obviously and nearly at the end of the module. On the other hand, the start of a module often does mysterious things to pointers and flags and may call other unknown subroutines for setup.

Second, be sure to use several passes through the tearing method. Pass one should deduce the structure and the overall use for each module in the code. Pass two should show you in fair detail more or less how the module works. Pass three should resolve any fine points or fuzzy details left over.

Here are some additional resources that make the tearing method even faster and more powerful . . .

Additional Tearing Resources
Apple IIe Reference Manual Apple Computer 20525 Mariani Avenue Cupertino CA 95014 (408) 966-1010 Beneath Apple DOS Quality Software 6660 Reseda Blvd Reseda CA 91355 (213) 344-6599 Copy II Plus Central Point Software Box 19730 Portland, OR 92719 (503) 244-5782 Disasm IIe Rak-Ware 41 Ralph Road West Orange, NJ 07052 (201) 325-1885

The *Apple IIe Reference Manual* is, of course, the technical manual for the IIe. This manual is *not* normally included with your Apple purchase and has to be ordered separately. It is extremely well written and well organized, and contains many details essential to understanding your Apple.

The book *Beneath Apple DOS* is another "must-have" reference. This one is especially important for tearing into programs that access DOS on the machine language level. Apple Writer IIe does this extensively for its file searches and super-fancy fast textfile manipulations.

There is also a sequel volume called *Bag of Tricks* which includes a diskette that is most useful in repairing and restoring "blown" disks.

I first bought the Copy II Plus program to speed up producing the companion diskettes for Enhance I. Besides being much faster than your usual copy programs, it verifies the results and easily lets you check your drive speed before starting any copy work. One most handy feature lets you rearrange files as you copy them.

Once I had this program in house, it became most useful for making backup copies of just about everything. The direct disk sector editor turns out to be very useful to solve all sorts of problems. We will use it later to let you add a POKE command to WPL.

The program DISASM IIe is an intelligent disassembler. This is one of the best available, although it is somewhat user vicious. In theory, you can pour object code into the top of a disassembler program and have source code pour out the bottom. In practice, there's lots of time and intervention involved, particularly with labels, comments, and documentation.

The single most important and most powerful use of an intelligent disassembler, though, lies in its *cross reference* abilities . . .

> Cross Reference—
>
> A detailed listing of each and every machine language location that is referred to or otherwise used by a program.
>
> Cross references often include separate listings of internal, external, and page zero values.

A cross reference listing is an extremely powerful way to check your progress in tearing apart any program. As three obvious examples, you can find out who uses known monitor locations for such things as outputting characters, clearing screens, or whacking the speaker.

Or, once you crack a subroutine, you can go back through the code and identify the use of that newly cracked subroutine everywhere it is called. Once you understand the use of a page zero pointer, flag, or stash, you can go back through the code and identify each and every use of this value everywhere in the program.

I call this the *avalanche effect* . . .

> Avalanche Effect —
>
> Using cross references to plug known results back into unknown portions of the code yet to be analyzed.

The reason that the avalanche effect is so potent is that it has gain. Just like a tiny snowball can totally destroy an entire mountain village.

For instance, five locations may call the Apple monitor COUT routine. Each of these five service subs may get called by five higher level routines. Start with COUT and avalanche it back to crack the service subs. Then avalanche the use of the service subs back into the higher level routines. From *one* monitor hook, you find the use and purpose of *30* different modules of your unknown code.

Say you are puzzling over a totally weird block of code that does heaven knows what. It is so gratifying when the avalanche effect suddenly plops a "GLOSSARY NESTING ERROR" screen message right onto one module exit. You now know the use and purpose of your undecipherable code, without any analysis at all!

The gain of the avalanche effect makes cross referencing a very powerful weapon for tearing into unknown code.

One important warning though. Be on the lookout for aliasing . . .

> Aliasing —
>
> Fake cross references that get created when you attempt to disassemble a file or some other nonworking part of your code.
>
> Aliasing also can happen if you begin disassembly in the middle of an otherwise legal op code.

Remember that *any* disassembler, intelligent or not, always assumes that it is working on legal code starting from the beginning of a legal 6502 instruction. If you try to disassemble files, you will get fake cross references created that do not exist.

For instance, say you have a low ASCII character file with a space, followed by an "A", followed by a "B". The disassembler will say "Ah! A $20, then a $41, then a $42." The disassembler will assume this is a legal 6502 instruction, and disassemble it as a JSR $4241. Naturally, the $4241 will show up on your cross reference list. The $4241 is an alias, since the code has nothing at all to do with whatever the *real* $4241 location happens to be up to.

As a more subtle example, you often may end up missing the starting point of a string of legal op codes. If this happens, the first few op codes will get messed up, but the disassembly will eventually straighten itself out. Once again, some aliased and nonexistent cross-references may get created.

Not to mention some really dumb instructions doing some really weird things.

The easiest way to avoid most aliasing is to never disassemble anything that you are not certain is working code. Should a few aliases creep in, you can edit them by hand to get rid of any remaining problem locations. Some special techniques, involving a table called SNEAKYSTUFF, will help us avoid aliasing when we capture AWIIe source code later on.

ANALYZING APPLE WRITER IIe

One or two warnings before we begin. Everything here is unofficial and unsanctioned. What you see here is just my use of my tearing method on my copy of my "F" version of the program. I've avoided using labels here since they are certain to conflict with the "real" labels on the "real" source code. In some cases, I've had to make a guess or two in understanding fuzzy parts of the code. In other place I have missed something obvious, and in yet others I could easily be just plain wrong.

Secondly, the tearing applies to the "F" version of *Apple Writer IIe*, circa March of 1983. Any changes at all to the program, or any use of a different version will change many of the program locations.

So, use what you see here as a guideline towards tearing apart your own version of most any old word processor of your choosing. But, do not accept anything you see here as gospel, because it is not. By all means, use the hotline to close the loop on any updates and corrections.

There is one thing that leaps out at you when you tear into this program: Here is a program with class, that was written by a master that knows and loves his Apple, and who programs by one-on-one interacting with his machine on his own terms.

I've *never* seen, used, or torn apart *any* other program that exudes such great vibes.

Another thing that becomes obvious when you tear into this program is seeing the delicate balance involved in designing a major word processor. People want a word processor that is cheap, has a large work space, is very fast, is powerful, is easy to learn, and has many different features. All of these demands continually fight each other six ways from Sunday.

How do you analyze a really big program?

Well, first you use the tearing method to find out everything you possibly can about it, color coding with the highliters as you go along. Then you use a cross-reference and the avalanche effect to nail down any loose ends. And finally, there it sits, a completely documented and torn listing.

But you still may not understand it. Now what?

Here is one way to analyze a heavy program . . .

To Analyze a Heavy Program —

(0) Use the program until you know it cold and have *completely* mastered its operation.

(1) Tear the program apart.
(2) Study the memory map.
(3) Find out how each file works.
(4) Learn all uses of page zero.
(5) Master the low-level subs.
(6) Attack high-level entry points.
(7) Try simple mods.
(8) Capture the source code.

Step zero is far and away the most important. There is no possible way you can understand *any* program if you do not use the program daily and continuously until you *completely* understand what the program does and how it does it.

Omit step zero, and nothing that follows will make any sense at all.

The best way to understand the other steps is to do them in order. We'll only briefly touch on the booting process here. As a reminder, we are assuming an Apple IIe with 128K split as a main 64K RAM bank and an auxiliary 64K RAM bank plugged into slot zero. We also assume you are under DOS 3.3e.

On a boot, a slightly modified version of DOS 3.3e is installed into high main RAM. This DOS creates standard ASCII text files that are totally readable by normal DOS 3.3 or DOS 3.3e.

The DOS boot process loads and runs a machine language program called OBJ.BOOT. OBJ.BOOT, in turn, analyzes the machine to see what is connected where, and initializes a few locations and flags. If you do not have an Apple IIe, an error message program called OBJ.APWRT][D is run that tells you the bad news, hangs the machine, and then kicks sand in your face.

If you have *no* 64K memory card in slot zero, a machine language program called OBJ.APWRT][E is then loaded and run. This "E" version gives you only 27K of main text, besides cramming the machine to the gills.

Note that the "E" version is slightly shorter, and significantly different from the "F" version we are going to analyze here.

If you have the 64K extended memory card in slot zero, a *different* machine language program called OBJ.APWRT][F is booted, starting at location hex $2300.

This is the "F" version we will study. The "F" version gives you 48K worth of main text file, and has lots of space left over for your customizing.

Both the "F" and "E" versions are greatly improved offspring of the old Apple Writer 2.0, which in turn was a major overhaul of the original Apple Writer versions 1.0 and 1.1. If you have torn apart all of these, it sure is interesting seeing evolution in action.

The "E" or "F" modules are easily loaded and analyzed by ordinary DOS 3.3 or 3.3e. Everything is up front. No sneakiness or black magic is involved. If you want to view, list, modify, or even capture the program under your own source code, a simple BLOAD OBJ.APWRT][F, A$2300 does the job. More details on this later.

THE MEMORY MAPS

The biggest step towards understanding heavy code is to find out what sits where in the machine. One or more *memory* maps does the trick . . .

> Memory Map —
>
> A picture or graph that shows you what portions of the Apple IIe address space are used for what purposes.

If you do not know where everything sits in the machine, or understand how the various parts are intended to work together, then there is no hope of going any further.

Getting a memory map with exactly the right amount of detail is often a real hassle. I like to use both *simplified* memory maps that show only the big picture, and separate *detailed* memory maps or lists that give you all the gory details down to every use of every last bit.

Fig. 12-1 shows us the simplified memory map for the "F" version of Apple Writer IIe. As a reminder, this is a 128K machine split into a main RAM bank of 64K and an auxiliary RAM bank of 64K, and we are under a somewhat modified DOS 3.3e disk operating system.

No use is made of any ROM in the machine, except during setup. A copy of the high end of the monitor ROM gets moved to high main RAM from $F800 through $FFFF.

As we'll see later on, very few monitor routines are used at all. Those that are get accessed from a clone in RAM. All of the 80-column screen routines are also built into Apple Writer IIe. These routines are faster and more powerful than the stock ones. No "escape" cursor movements are needed since the program uses a continuously live screen supported by full cursor control keys.

As is typical, 4K main memory bank "2" is used; bank "1" is not.

DOS 3.3e is loaded during the booting into high main RAM just below the monitor clone. Note that this is a non standard location some $3800 bytes higher than usual. One reason this relocation was done was to leave enough room on the "E" version for a decent text file area.

The actual word processing program code is also loaded into main RAM. The working code sits between $2300 and $4AF9. Some reference files that consist mainly of screen messages and address pointer tables are towed along and follow the working code between $4AF9 and $53D0.

Many work files are needed by this program. To name a few, these include the glossary buffer, two deletion buffers, and the WPL program storage area. There are many others, as we will see later. The work files are stashed in main RAM from $0800 through $2300.

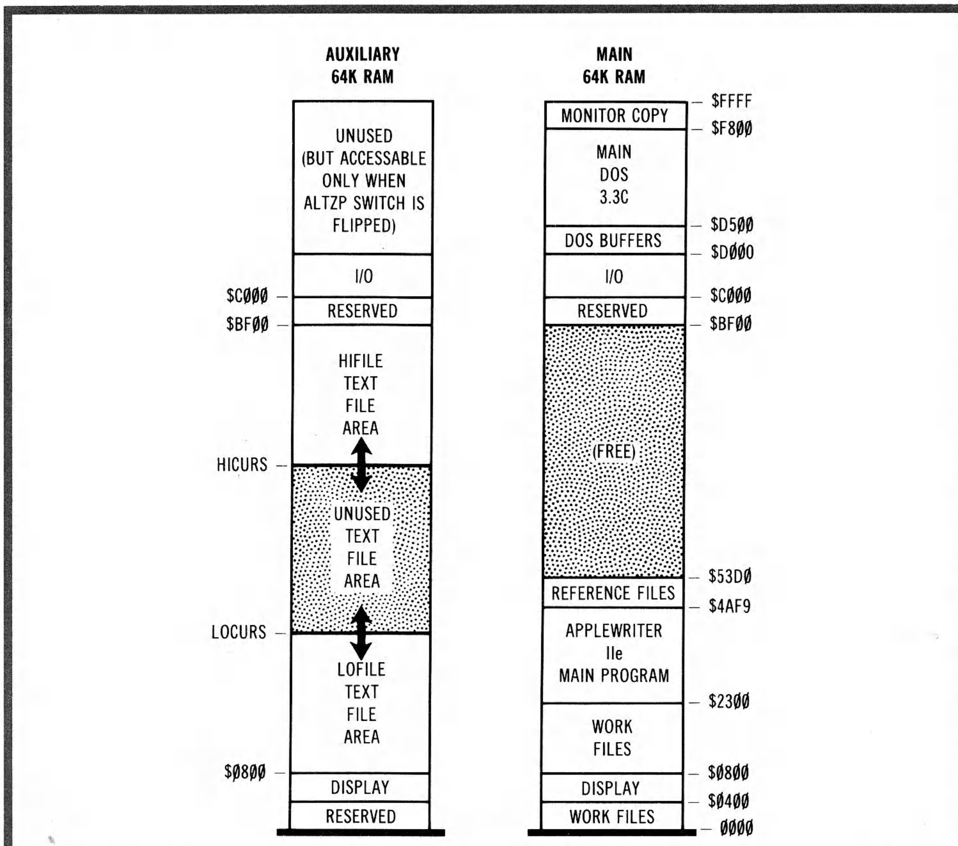


FIGURE 12-1. Simplified memory map of 128K Apple IIe running the "F" version of Apple Writer IIe.

The main RAM area from $0000 through $0400 is also used as a work file area. This includes the important pointers, counters, stashes, and flags on page zero; some memory management and a stack on page one; and some additional work files and links on pages two and three.

Most of the work files are *not* loaded into the machine. They are initialized and then used as the program is run.

As usual, the *even* 80-column characters are stashed in main RAM from $0400 through $07FF, while the *odd* 80-column characters are stashed in auxiliary RAM in the same address range. Also as usual, $C000 through $CFFF is reserved for I / O space uses.

A humongous chunk of unused RAM sits between $53D0 and $BEFF in main RAM. It is very lonely and is crying for your attention and use. What can you cram into 27439 bytes these days?

Turning to the 64K auxiliary RAM, the auxiliary page zero and stack area are not used. Since high auxiliary RAM is switched with page zero, all 16K of the high auxiliary RAM is also unused.

Messing with the alternate page zero and alternate high RAM gets tricky fast. It turns out that there is an attempt in this program to write an unneeded and unusable DOS clone into high auxiliary RAM. The attempt fails. What really happens is that the DOS in main high RAM gets copied back over itself.

No harm done.

The text file area that holds the words you want to process takes up the bulk of auxiliary RAM. Your text file area goes from $0800 through $BEFF, and gives you room for some 46,845 characters at once.

As with main RAM, auxiliary RAM locations $BF00 through BFFF are reserved for system globals, as might be needed for fancy memory management.

Generally what happens is this: The main program puts characters into or removes them from the text file area, providing all the usual word processing functions as this happens. It gets these characters from you at the keyboard, from DOS as text files, from the text file itself for clones and copies, or out of special files such as the glossary, the WPL program, or the deletion buffers. When finished, the main program saves what is in text file area to disk or dumps it to a printer.

So much for the main memory map. We will be giving you two levels of additional detail on most of these areas. In the text that follows, we will tell you generally what each area is up to. In the various tables, you will find the extreme detail needed for a complete analysis.

Time now to look at . . .

HOW EACH FILE WORKS

There are many different files used in this program. Dozens upon dozens of them. Once you find where a file sits and what it does, you are well on your way to understanding just how Apple Writer IIe does its various tasks.

Let's break up the files into four areas, depending on what they do. These are the *text file* area, the *work file area*, the *internal file* area, and the *reference file* area.

The text file holds the words to be processed. As we will shortly see, this is actually a pair of files, done so to dramatically speed up character insertion and deletion. The work files hold things outside of the program code that are fairly likely to be changed. This includes all the page zero stuff, the glossary, print programming commands, the tab file, top and bottom line buffers, and much more.

The internal file area is more or less an afterthought. These are files that are stuffed inside the working code, or are otherwise "misplaced." Most often, these are involved with an improvement or upgrade of the older Apple Writer programs. The TAB status line display file is typical. Putting files any old place is admittedly a little sloppy. But, its a great reminder that this was written by a person, rather than by a committee or a machine.

You have to pay very close attention to these internal files since you will want to bypass them when you capture your own source code. Otherwise, you will get aliasing and "starting off on the wrong foot" problems.

The reference file area holds things that are unlikely to change. These include screen prompts, address pointers, error messages, DOS commands, and so on.

A summary . . .

Apple Writer IIe File Areas
Text File Area —
Holds the words to be processed.
Work File Area —
Holds things often changed.
Internal File Area —
Uh, whoops.
Reference File Area —
Holds things seldom changed.

Let's check into these file areas one at a time. The single largest and most important file area is the . . .

Text File Area

The text file area holds the words to be processed. This area lies in auxiliary RAM from $0800 through $BEFF, a total of 46,847 locations. Allowing for the two $FF end markers leaves you with a remainder of 46,844 characters. This is the number you see as the MEM prompt on program bootup.

Fig. 12-2 shows us how this text file area is managed.

One very sticky problem in word processing involves making your code run fast enough that it never gets very far behind your typing. Say you have a single, long text file and are sitting in the middle of it. On each key entry, you would have to move everything from where you are sitting up or down a character in memory. This could involve tens of thousands of characters and is bound to be ridiculously slow.

The trick to fast word processing programs is to not move any characters that are already sitting in memory during any fast typing modes . . .

> The secret to a fast word processing program —
>
> NEVER move any character that is already sitting in memory, during any fast typing mode.

Sounds fair enough.

But ignoring this key rule is what makes so many competing programs so abysmally bad.

Now for the key secret to Apple Writer IIe, and the brilliant way to avoid moving things around all the time. The way around this problem is to use *two* separate text file areas, keeping the available free space remaining *between* the two files!

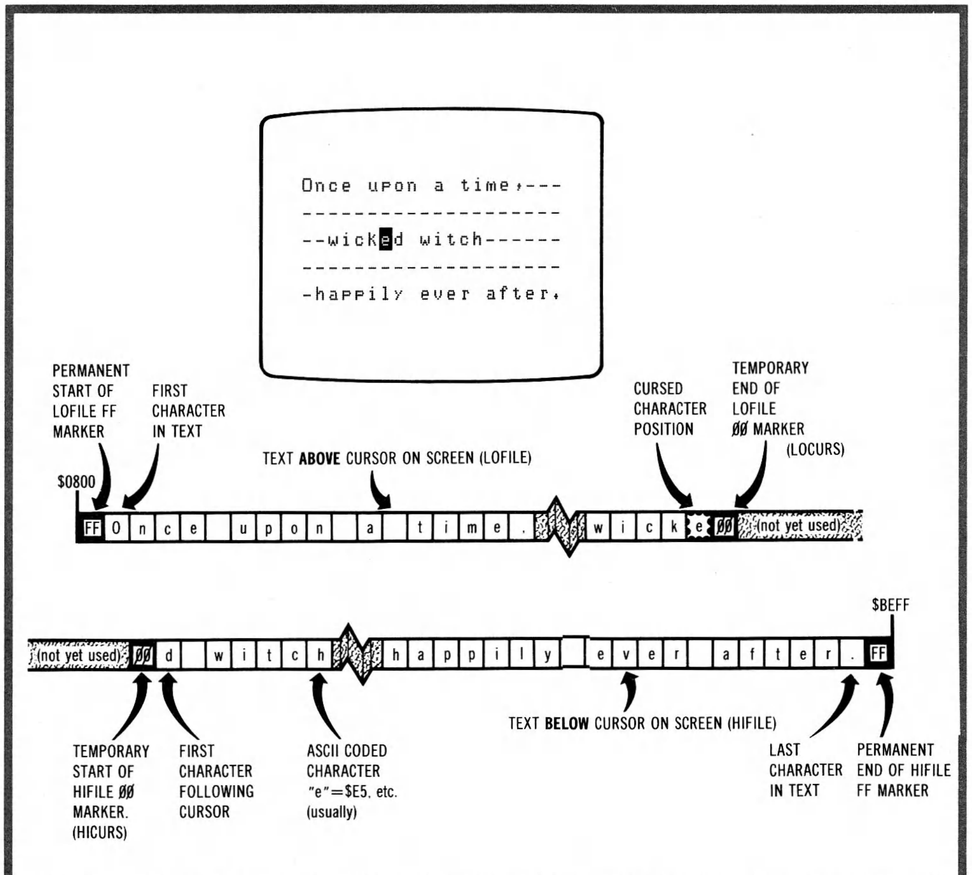


FIGURE 12-2. The Apple Writer IIe screen often displays two files. The LOFILE in RAM holds the characters above the cursor. The HIFILE in RAM holds the characters below the cursor.

Turning to Fig. 12-2, we see that there is a file I call LOFILE that holds everything *above* the cursor on the screen back to the beginning of the text. There is a file I call HIFILE that holds everything *below* the cursor on the screen on out to the end of the text.

During normal character entry *or* insertion, you simply add things to the end of LOFILE. Since HIFILE is usually far above LOFILE in RAM, there is no difference between insertion and entry. To delete, you simply knock characters off the top of LOFILE, again without disturbing HIFILE. Nothing but a single character need be entered or removed from the file for most fast typing needs.

The files do get back together every now and then. For instance, on an [E] command to go to the end of the text, everything gets moved back into LOFILE. Your *entire* file now starts at the beginning of the file, and you are free to add to the end of LOFILE with new characters.

It does take the better part of a second to move everything in a long text file from HIFILE to LOFILE, but you do this only rarely. It's unlikely that you would want to continuously type [E] commands at a 100 word per minute rate. In fact, you are usually ready for a brief psychic break when you do an [E]. This is human engineering at its very best.

Similiarly, if you do a [B] to get to the beginning of a file, everything gets moved to HIFILE. If you add characters from here, they go into LOFILE. Once again, nothing needs to be moved in memory during fast entry modes, even on an insertion or deletion.

Meanwhile, back at the screen, fancy things are going on. The screen shows you copies of pieces of HIFILE and LOFILE. It magically splices them together as needed to con you into thinking you are looking at one continuous file. Only whole words are shown in the wraparound mode.

The screen is usually updated on each character entry. But there are far fewer characters on the screen than are usually stashed in the text file area. So this screen updating can be done fairly fast. It does take somewhat longer to update insertions than additions.

A few users complain that AWIIe "misses" characters. Despite a 64-key type-ahead buffer, a rare miss can happen when the slower insertion mode and a subtle bug in the IIe keyboard encoder scanning gang up on a sloppy typist. To get around this, do as much typing in the entry mode as possible, and add some extra "snap" to the quick release of all keys. It is important to "let go" of an old key as soon as you can.

The *bottom* of LOFILE at $0800, and the *top* of HIFILE at $BEFF are always identified by $FF markers. These tell various service routines, such as the searches and finds, when they get to the beginning or the end of either LOFILE or HIFILE. The "open" end of each file is also marked with a $00 marker. This is done at the *top* of LOFILE and at the *bottom* of HIFILE.

The cursed character on the screen sits at one less than LOFILE, and is just below the LOFILE $00 marker.

The $FF "limit" markers and $00 "present end" markers are reserved characters. You are not allowed to enter these characters into your text file. Characters of $7F and $80 are also not permitted.

Note the extra zeros are allowed in the unused memory space before LOFILE and HIFILE. It is only the *first* zero on the way up through LOFILE, or the *first* zero on the way down through HIFILE that matters. If you want to eliminate a string of text you simply put a zero at one end. This saves having to "erase" lots of memory.

A cursor is produced by taking the top character in LOFILE and alterating its screen display between normal and inverse, following a software loop that causes apparent flashing. By the way, the "alternate" dual-case character set is used that does not have a hardware flasher available.

Most of the characters in the file are standard high ASCII. To simplify screen updates and word wraparound, the end of each screen line is set to a low ASCII character. This unique approach does wraparound calculations only once, rather than needing special treatment on every screen update. The low ASCII marker usually is a carriage return or the space following the last whole word that will fit the screen line.

Note that low ASCII characters have their most significant bit, or MSB, cleared to zero. High ASCII characters have their MSB set to one. High ASCII is used for most Apple uses most of the time. As a reminder, characters $00, $7F, $80, and $FF are reserved and must not be placed into a text file.

In certain ways, then, Apple Writer IIe is a line oriented word processor, because it remembers exactly what each screen display line should look like at all times.

When text files are saved to disk, all the saved characters are forced to high ASCII. Thus, Apple Writer IIe files are easily exchanged with any program on any computer that can recognize a standard text file full of ASCII characters.

Which is crucial for typesetting, fancy print formatting, and for modem communications.

Let's review.

The 47K text file area usually holds *two* files called LOFILE and HIFILE. LOFILE holds everything *above* the screen cursor and HIFILE holds everything *below* the screen cursor. These dual files are used to prevent having to move things around during insertions and deletions, and are the keys to acceptable word processing speed.

The beginning of your text at the bottom of LOFILE is marked with an $FF marker. The end of your text at the top of HIFILE is also marked with an $FF marker. The "open" ends of LOFILE and HIFLE are identified with $00 markers. These two open ends face each other, with all of the remaining memory space between them. The cursed character is the last one at the top of LOFILE. Text is added to the top of LOFILE during entry and insertion. During deletion, text at the top of LOFILE is replaced with $00 markers.

Every now and then, LOFILE and HIFILE are merged back together, such as with a [B] that puts everything in HIFILE or an [E] that puts everything back down in LOFILE. As one fills, the other empties. More often than not, everything is in LOFILE and you are adding to the *open* end of LOFILE during normal text entry.

Text is normally entered in high ASCII. All characters are allowed except for ASCII codes $00, $7F, $80, and $FF. A possible conflict with the delete key is handled by recording delete as $80, or as a NULL command in high ASCII. The end of each display screen line is held as a low ASCII character. This low ASCII marker provides automatic word wraparound and needs only a single calculation, rather than special processing on each and every screen update.

Note that you cannot enter an ASCII "do-nothing" or NULL command into a stock Apple Writer IIe textfile, since the $00 coding is reserved.

Be sure you thoroughly understand how this LOFILE and HIFILE pair operate, for they are the key to understanding the entire program and everything that follows here.

The next step in our file understanding involves . . .

The Work File Area

The work file area holds *nontext* values that are likely to change as the program is used. The work file area extends from $0000 through $22FF on the main page of RAM.

A detailed work file listing appears as Table 12-1.

Let's briefly see what these files are and what they do. Starting at the bottom, much of page zero is reserved to hold pointers, counters, stashes, and flags. These are so important and so crucial to your understanding how this program works, that we will reserve the next section just for page zero.

The area from $0100 through $0155 holds the memory management code. Note that main RAM page zero and page one are always used in this program. These do *not* change on a switch between main and auxiliary RAM. Only RAM memory locations from $0200–BFFF switch on a change from main to auxiliary RAM.

The memory management code lets you read the text file from a low cursor pointer I call LOCURS, a high cursor pointer I call HICURS, a screen update pointer, a printer pointer, or a general-use pointer.

The high end of page one holds the stack. As usual, the stack starts at $01FF and builds down. The stack is short enough that it never crashes down into the memory management code. The stack is used to hold subroutine return addresses and as an occasional temporary value stash.

Table 12-1. Apple Writer IIe Work File Details

The workfiles for Apple Writer IIe sit in main RAM from $0000 through $22FF. These are storage areas that are likely to change as the program is used.

Here are details on the work files:

$0000–00FF — **PAGE ZERO WORK AREA**

Page zero holds pointers, counters, stashes, and flags for program use. This area is so important that we have set aside Table 12-4 and Table 12-5 for full details.

$0100–0156 — **MEMORY MANAGEMENT CODE**

Routines to manage auxiliary memory are put here since pages zero and one are never switched or made inactive in this program.

There are eight management routines. These get installed during the cold start:

$0100–0108	—	Handle a DOS error via main memory.
*$0109–0118	—	Send a character to the DOS clone in auxiliary memory.
$0129–0131	—	Read screen character from auxiliary memory.
$0132–013A	—	Read cursed character from top of LOFILE.
$013B–0143	—	Read character from bottom of HIFILE
$0144–014C	—	Read character to be printed from LOFILE.
$014D–0156	—	Read character pointed to by general use pointer.

(* doesn't really happen — see text)

$01??–01FF — **6502 STACK**

The 6502 stack inits to $01FF and builds down. The stack is used to save the return address on subroutines and separately to temporarily save and restore register values. Since the stack builds down, the area below the stack is risky to use, although values between $0157 and 0180 are probably safe.

$0200–027F — **KEYBUFFER**

Keystrokes are read into this keybuffer, either directly from the keyboard or else from the type-ahead buffer during hectic times. This is a major work area where such things as delimiters are processed. ASCII characters start at $0200 and build up in memory, ending with an $8D carriage return.

$0280–02DF — **ACTIVE FILENAME BUFFER**

This area holds the filename in active use for DOS access. The "=" filename is saved separately in the work file at $1A40–1A7F. ASCII characters start at $0280 and build up in memory. A $00 value at $0280 means that a legal filename is no longer there. The end of the file is also filled with zeros to prevent any mixups on slot or drive. The boot process inits this file to a $00 at $0280, as does the DOS error handler.

$02E0–02FF — **DOS INPUT/OUTPUT BLOCK**

The DOS input-output block, or IOB, sits here. Complete use details appear in Beneath Apple DOS. On a machine language call, DOS goes here to find out what it is supposed to do.

Important IOB locations include:

$02E1	—	Slot number $01-02
$02E3	—	Volume number
$02E4	—	Track $00–23
$02E5	—	Sector $00–0F
$02E8	—	Buffer address low
$02E9	—	Buffer address high
$02EC	—	Command 1 = READ, 2 = WRITE

$0300–037F — **CHARACTER SWALLOW BUFFER**

Single characters being deleted get saved here by the open-apple, left arrow command. They are restored by the open-apple, right arrow command. The buffer is 128 characters long and works on a round-and-round basis. A pointer at $AC decides where to put or get the next character. Each ASCII character is put one address higher than the previous one.

$0380–03CF — **APPARENTLY UNUSED LOCATIONS**

$03D0–03FF — **SYSTEM VECTORS**

Addresses of key DOS, interrupt, control, and reset vectors normally sit in this area.

Table 12-1—cont. Apple Writer IIe Work File Details

Of interest here are:

$03D2	— Starting page of resident DOS usually $D5 for DOS installed in high RAM $D500–F7FF.
$03D6	— Jump code to the DOS File Manager subroutine.
$03D9	— Jump code to the DOS RWTS sub for reading or writing to tracks and sectors per IOB instructions.
$03E3	— A subroutine to find the DOS input parameter list for RWTS.
$03EA	— Jump code to the subroutine that reconnects DOS to the I/O hooks.
$03F5	— DOS error hook to get back to the main error processor at $480B.

$0400–07FF — **TEXT SCREEN**

Characters to appear on the screen are mapped into this memory area. The even characters go in main memory and the odd characters go in auxiliary memory for the 80-column screen.

Note that all screen code is done inside Apple Writer IIe. The usual monitor routines are not used since they are slow, have memory conflicts, do not save keystrokes, and do not handle screen motions in the way needed.

$0800–0BFF — **WORD AND PARAGRAPH DELETION BUFFER**

Whole words and entire paragraphs are saved and restored to this area with the [W] and [X] commands. A pointer pair at $94 and $95 continuously points to the next available location. This is done on a round and round basis, with the character after $0BFF going into $0800. A separate counter pair of $EF and $F0 keeps track of >1024 overflows. A space ends [W], while a carriage return ends [X]. The boot process inits this work file to all carriage returns.

$0C00–0CFF — **DOS TRACK AND SECTOR BUFFER**

This 256 byte buffer is used to hold a DOS T/S, or track and sector list for direct machine language loading of text files. Direct access is both faster and lets you search for portions of a complete text file. A pointer at $F1 is used to access this buffer, which is filled by DOS itself.

$0D00–0DFF — **DOS TEXT FILE BUFFER**

This 256 byte buffer is used to hold part of a text file read by DOS under RWTS. Each piece of the text file can be searched as needed for delimiters. A pointer at $F0 is used to access this area, which is filled by DOS itself.

$0E00–15FF — **WPL PROGRAM FILE**

A WPL program to be run goes here. Each WPL command consists of a group of high ASCII characters ending with a carriage return. A $00 value marks the end of the program in case the QT command is missed. The WPL program counter $A0–A1 reads this file, controlled by the WPL continue flag at $E7 and the WPL activity flag at $DF. In use, each WPL statement is read, interpreted, and then carried out. The WPL program file can be 2048 characters long if no footnotes are in use. With footnotes, the file can only be 1024 characters long and must end at $1FFF.

$1200–15FF — **FOOTNOTE BUFFER or WPL PROGRAM FILE**

If footnotes are in use, they are held here from the time the footnote occurs in the text until the bottom of the current page being printed. The ASCII characters build up from $1200 with each separate footnote ending in a carriage return. A $00 marks the end of the last footnote. Flag $FE keeps track of footnote use, with pointer pair $00,01 locally used to load and then read this file. If footnotes are not in use, then these 1024 locations can be used as additional WPL work file memory.

$1600–16EF — **LINE JUSTIFY BUFFER**

These 240 locations are used to format a line being justified, as well as to hold the searching delimiters during [L]oad, and to handle word wraparound during screen line formatting.

$16F0–$16F5 — **DECIMAL STASH**

Holds a decimal number expressed in ASCII, along with a possible sign bit. Used for hex-to-decimal and decimal-to hex conversion, with the corresponding hex value sitting in $C0 and $C1. This work area is inited to all zeros before use.

$16F6–16FF — **APPARENTLY UNUSED LOCATIONS**

$1700–173F — **WPL STACK**

These 64 locations hold the 32 possible subroutine return addresses for SR commands in WPL. Pointer $92 accesses these locations a pair at a time, entering a new address pair on each SR and reading a pair on each RT return.

$1740–177F — **TYPE-AHEAD CHARACTER BUFFER**

The 64 locations here hold characters between the time they are typed and the time they can be used, allowing the user to get as many as 64 keys ahead of the program without any errors. A filling pointer $F3 enters the characters as they are typed. An emptying pointer $F2 gets the characters as they can be used. During nonhectic times, $F2 = $F3 and the buffer is empty. The buffer goes round and round, with the next key after $177F going into $1740. Should an open-apple or a closed-apple key be used with a normal key, these are separately saved in the apple key buffer at $1AC0—1AFF.

Table 12-1—cont. Apple Writer IIe Work File Details

$1780–17BF — **WPL CHARACTER STRING $A**

$17C0–17FF — **WPL CHARACTER STRING $B**

$1800–183F — **WPL CHARACTER STRING $C**

$1840–187F — **WPL CHARACTER STRING $D**

These four work files hold the WPL character strings $A-$D. Each string consists of high ASCII characters building up from the starting address and ending with a $00 marker. A $00 at the starting address means this string is not yet in use. All four strings are inited to $00 during the boot process. Pointer $8E acts as a locator to read these strings.

$1880–18BF — **TAB FILE**

These 64 locations hold up to 32 different tab values. Tabs past 256 characters into a paragraph or complicated form are allowed. The pointer pair $96,97 keeps track of tab positions into a paragraph. On a tab purge, the entire file is filled with zeros. On a tab set, the pointer value is stored in the first nonzero location. On a tab clear, one pair of locations is zeroed.

$18C0–19FF — **PRINT PROGRAM FILE**

The print program file holds the top line stash, the bottom line stash, and the individual print constants. These are loaded on a[Q]-C and saved on a [Q]-D. Details follow below.

$18C0–193F — **TOP LINE STASH**

$1940–19BF — **BOTTOM LINE STASH**

The top line and bottom line are held here respectively in unformatted form, exactly as they appear on the PP screen. A $00 value goes at the end of the line. If no TL or BL is to be used, a $00 starts the respective file.

$19C0–19FF — **INDIVIDUAL PRINT/PROGRAM VALUES**

This stash holds all the individual print program values as needed by print formatting and WPL. Two bytes are reserved for each value, even where small numbers are normally used. This simplifies entry and processing. When an 8-bit value is expected, the first, or even byte is used and the second ignored. Here's a breakdown:

$19C0–19C1 — Left margin LM
$19C2–19C3 — Paragraph margin PM
$19C4–19C5 — Right margin RM
$19C6–19C7 — Top margin TM
$19C8–19C9 — Bottom margin BM

$19CA–19CB — Page number PN
$19CC–19CD — Printed lines PL
$19CE–19CF — Page interval PI
$19D0–19D1 — Line interval LI
$19D2–19D3 — Single spacing SP

$19D4–19D5 — Print destination PD
$19D6–19D7 — WPL numeric (X)
$19D8–19D9 — WPL numeric (Y)
$19DA–19DB — WPL numeric (Z)
$19DC–19DD — Carriage returns CR

$19DE–19DF — Underline token UT
$19E0–19E1 — Justify mode:

$00 = FJ
$01 = LJ
$02 = RJ
$03 = CJ

$19E2–19FF — These 30 locations are unused but are conveniently loaded and saved as PP values. The PRT.SYS seems to contain a fragment of some assembly code here.

$1A00–1AFF — **TOP AND BOTTOM LINE FORMATTER BUFFER**

This page of memory has several separate and local uses. Since each use is temporary, several different tasks can be done without conflict.

During print formatting, these 256 locations are used to expand the top display line or the bottom display line from its PP form using delimiters to its "open" form with full space padding and a real page number.

Table 12-1—cont. **Apple Writer IIe Work File Details**

$1A00-1A3F	—	**MULTIPLE USE BUFFER** There are three local uses for this buffer area. It is used to create DOS filenames for the PRT and TAB files. It is used to append slot and drive changes onto existing DOS files. It is used as a workbuffer when substituting for WPL (X), (Y), or (Z) numeric values.
$1A40–1A7F	—	**DOS FILENAME SAVE** The "=" filename gets saved here during DOS access that does not require the main filename. Used by [G]lossary, [L]oad (while appending), [S]ave, and [P]-DO. The "=" filename is replaced to the active filename buffer $0280–02DF after each DOS access, freeing this area up for T1/BL formatting.
$1A80–1ABF	—	**FIND STRING SAVE** The "=" search and replace string gets saved here for possible reuse. Using [F] = will repeat the previous search.
$1AC0–1AFF	—	**APPLE KEY BUFFER** An auxiliary to the main type-ahead buffer at $1740-17FF. This area remembers whether an open-apple or a closed-apple key was also pressed at the same time another key was pressed. Flag $FB holds this status on reading the type-ahead buffer pair.
$1B00–23FF	—	**GLOSSARY FILE** All of the glossary entries go into this file area. High ASCII characters build from the bottom up and each entry ends with a carriage return. Fake carriage returns are stashed with a "]" character if they are needed inside a string. A $00 marks the end of the last string. Both the boot process and the purge command "empty" this file by putting a $00 at $1B00.

The keybuffer extends from $0200 through $027F and holds string values and key commands that are entered from the keyboard. The keybuffer is also sometimes used as a temporary work area or to hold an old character string for later reuse.

An active filename hold sits just above the keyboard buffer. The DOS input-output block, or IOB, follows, starting at $02E0. This block is used for machine language access to the DOS code involved in RWTS, or reading and writing to individual tracks and sectors. Full details on RWTS appear in *Beneath Apple DOS*.

Important locations in this IOB include the track number at $02E4, the sector number at $02E5, and the read ($01) or write ($02) command at $02EC.

Continuing upwards through the work files in main RAM, the bottom half of page three is used for the single character swallow buffer, controlled by the right or left arrow key in combination with the open apple key. Note that the swallow buffer is separate from the combined word and paragraph deletion buffer higher in the work-space.

The usual hooks appear on the top of page three, starting at $3D0. Most hooks vector to a warm restart of the program. The ampersand hook is used for disk error recovery and vectors to a DOS error or message handler.

We'll breeze on by pages four through seven, since these are the *even* characters of the text screen. The companion *odd* characters of the text screen are stashed on pages four through seven of the auxiliary memory.

There are 1024 locations ranging from $0800 through $0BFF that are set aside for the word and paragraph deletion buffers, activated by [W] and [X]. On a deletion, the stuff to be saved gets tacked onto the end of anything previously saved. On restoration, the stuff gotten back gets read until a space or a carriage return is found. The pointers to this deletion buffer go round and round. Overflow is kept track of by a separate counter on page zero.

Two DOS buffers are provided. The first starts at $0C00 and usually holds a T/S track and sector list. The text file read from disk is stashed, one page at a time, at $0D00. Direct buffer access during text file reads lets you scan the file for starting and ending delimiters, besides being much faster.

The WPL program buffer starts at $0E00 and holds any program commands needed when you are using WPL. You are allowed 1024 characters in the program if you are using footnotes. If you are not using footnotes, you are instead allowed 2048 characters in this file. A common area from $1200 through $1BFF holds your choice of footnotes being processed, or else more WPL commands.

Most of the single page from $1600 through $16EF is set up as a line justify buffer. This page is separately used to format the top and bottom lines as well as being useful during search and replace activities. The area from $16F0 through $16F4 is reserved as a workspace for decimal-to-hex and hex-to-decimal conversions. The decimal ASCII values are stashed in this work file area.

All of which brings us up to $1700.

Next is the WPL stack that resides between $1700 through $173F. This stack holds return addresses for any WPL subroutines in use. A total of 32 subroutine calls are allowed.

The type-ahead buffer follows. This area saves up to 64 previous keystrokes when the typist gets ahead of any processing. There are two pointers that access this file on a round-and-round basis. One fills, the other empties. Any open-apple or closed-apple key commands are separately saved in a separate 64 character buffer higher in the work file area.

Note that you must save both the key pressed and the state of the "<open apple>" and "<closed apple>" keys for a later recovery. Otherwise, certain key combinations will do the wrong things.

The four WPL strings of $A through $D are stashed next. Each can be up to 64 characters long.

The tab file is also 64 characters long and begins at $1880. Since tabs beyond 255 characters into a paragraph are allowed, each tab value takes two bytes. There is room in this file for 32 active tabs.

The Print/Program file values are next in line, and take up the area from $18C0 through $1BFF. First provided are two 128 character buffers for the top line and the bottom line. The top and bottom line are held in their "compact" form here, with delimiters and a "#" for possible page numbers. When formatted, the top line or the bottom line is expanded as needed into a multiple use buffer at $1A00.

These compact line buffers are followed by individual stashes of all the print values such as the page number, the justify mode, and so on.

Locations of the individual print/program values are detailed in Table 12-1. Our intent here is to get a look at the big picture. The tables have more detail for you, if and when you need it.

The area from $1A00–1AFF has several uses. Sharing is possible since each use is temporary. The entire buffer is used to format the top or bottom line into its expanded form with real page numbers and full space padding.

Other uses split this page into four temporary work files, starting with a short multiple use buffer at $1A00–$1A3F. This one gets used in assembling PRT and TAB filenames, to hold slot and drive values, and as a WPL substitution workbuffer.

An "=" filename buffer follows from $1A40–1A7F. This is used to hold the old text filename while DOS is being temporarily used for something else. For instance, the name of a glossary or a WPL file to be loaded may need the active filename buffer at $0280. During such special use, the old filename is briefly saved here. Immediately after special use, the old filename gets returned to the active buffer at $0280.

This is followed by an "=" buffer at $1A80–1ABF used for repeat searches and replaces. Next is an open- or closed-apple stash at $1AC0–1AFF used as the second half of the type-ahead buffer.

Finally, the area from $1C00 through $22FF is reserved for the glossary buffer. This sets aside up to 2048 characters for user-defined keystrokes. Each glossary entry ends with a carriage return. The final entry and return is followed by a $00 marker.

As you can see, there are lots of work files that are needed by Apple Writer IIe. These work files hold things likely to change during your use of the program. Much of the power and uniqueness of Apple Writer IIe comes about through extensive use of these work files. (Fig. 12-3).

That just about completes our survey of the work files. Next, let's briefly look at . . .

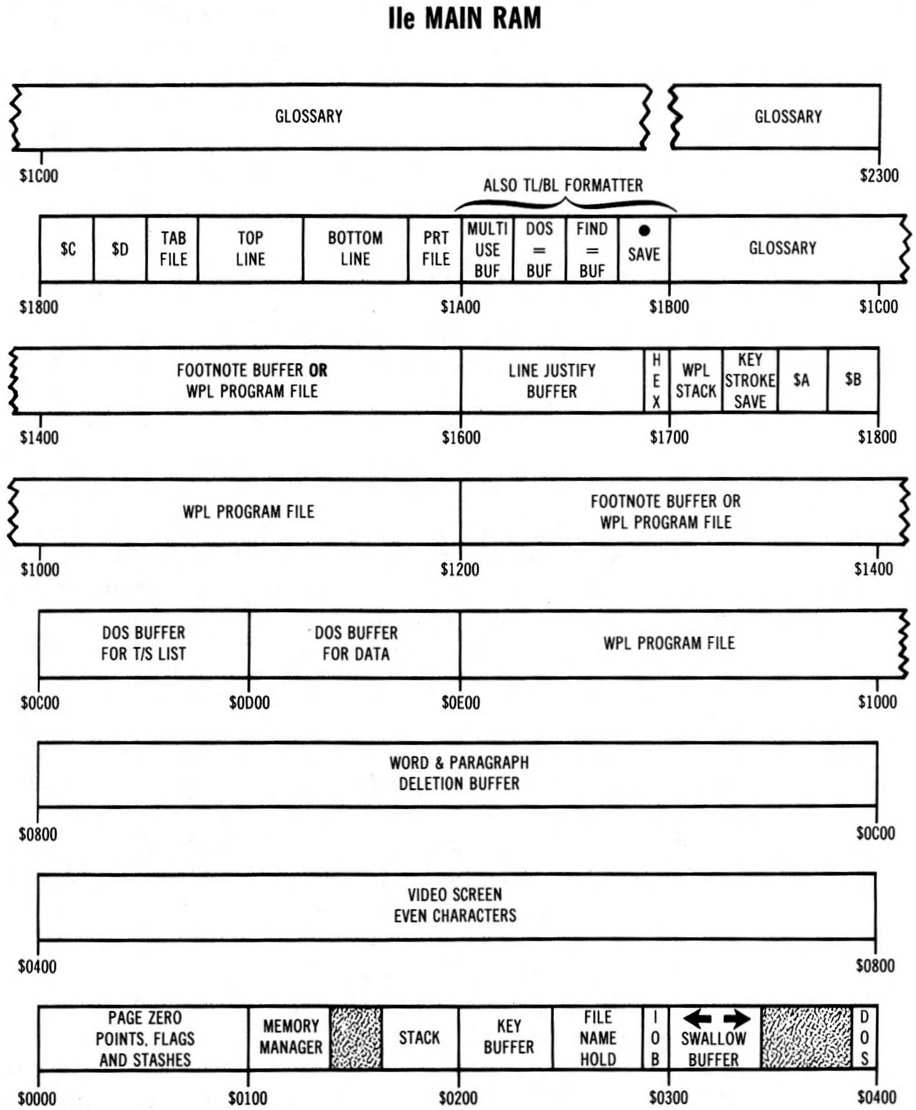


FIGURE 12-3. **Simplified memory map of Apple Writer IIe work files.**

Table 12-2. Apple Writer IIe Internal File Details

Internal files are "loose ends" that tend to accumulate where you don't want them to when you keep revising a program. These are stashes or images inside the main portion of the working code. These stashes and images must be patched around when you are capturing source code, or aliasing is almost certain to result.

Here are the internal files:

$26FE — **END OF LINE Y-SAVE STASH**

A one-byte location used to preserve the Y-register during the clear-to-end-of-line service subroutine.

$2B92 — **GLOSSARY NEST STACK POINTER**

Holds a pointer that, when doubled, points to address pairs in the following table.

$2B93–2BA2 — **GLOSSARY NEST STACK**

Holds eight pairs of addresses pointing to the glossary work file. When one glossary entry calls another, the return address is held here. This neat and very powerful code works just like a subroutine.

$3045–3095 — **TAB STATUS LINE IMAGE**

This ASCII file holds the image of the tab status display and is used to remember set tab values.

WARNING: This stash will alias bady if you try to disassemble or cross-list it! It will also alias and louse up the next two machine language instructions that follow it!

$3885–3886 — **FIND REGISTER STASH**

These two bytes are used to save register values during [F]ind. The X-register gets saved in $3885. The Y-Register is saved in $3886.

$47E0 — **PRINT MODE CHARACTER SAVE**

Holds a character during printing for the underline check. If the underline symbol is present, substitutes a space.

NOTE: Another "misplaced" stash involves $5210–5213 in the reference file area. See Table 12-3 for details.

The Internal File Area

The internal file area holds a few oddball things stuffed into the working code. This was done to handle updates and improvements without causing reassembly hassles. Table 12-2 shows us all the internal work files.

There are some X and Y register temporary stashes, the glossary nest, and the tab status line display. The glossary nest holds the addresses to linked parts of the glossary. More details appear in Table 12-3 and in the upcoming Table 12-7 script.

Our main interest in the internal file area is to let us bypass these locations when capturing your own source code. In particular, the disassembly process gets messed up coming out of the tab status display. What is aliased as "LDX $FF20, ROL $20, and $27" is correctly disassembled as "JSR $26FF, JSR $2725".

A lot of aliased references to $AEAE, $AEB3, etc., are also created if you try to disassemble the tab status file as legal op codes.

The final area we are interested in is . . .

The Reference File Area

The reference files hold things that are more or less permanent and that do not normally change much during program use. The reference files are stashed in main RAM from $4AF9 through $53CF.

Table 12-3 gives you more detail on these reference files.

We start with a screen base address table. Because of some hardware considerations, there is no obvious relation between where a character goes on the Apple

Table 12-3. Apple Writer IIe Reference File Details

The reference file area holds values that are seldom, if ever, changed. This file area sits above the main program from $4AF9 through $5400. See Table 12-6 for specific address location vectors.

Here are details:

$4AF9–4B28 — **SCREEN BASE ADDRESS TABLE**

Holds pairs of addresses corresponding to the leftmost base address of each screen character line. Thus, top line zero has a base address of $0400 and line one an address of $0480. Table lookup of screen base values is much faster than calculation.

$4B29–4C12 — **FUNCTION LIST**

An ASCII list of all available control commands, along with any bottom line prompts needed for those commands. Note that [@] is the delete key and that [A] is reserved, presumably for modem use. [M] is a carriage return. This table is scanned for a match character between the "[" and "]". If a match is found, a bottom line prompt is included on certain commands.

$4C13–4C44 — **FUNCTION ADDRESS LIST**

Holds the module addresses for the control commands [@] through [Z]. Thus, the [@] for delete module starts at $2AD0, [B] for move cursor to beginning module starts at $28DC, and so on down the list.

$4C45–4C6C — **PRINT CONSTANTS MATCH FILE**

A file of pairs of ASCII characters from LM, PM, ... through RJ, and finally CJ. On a match to a character pair, the entered value is saved to the proper PP slot.

$4C6D–4C91 — **WPL COMMAND MATCH FILE**

A file of pairs of ASCII characters from GO, DO, ... through SI, and finally EP. On a match to a character pair, a jump is done to the module that handles that WPL command.

$4C92–4CB5 — **WPL COMMAND ADDRESS FILE**

These address pairs hold the starting point of the WPL modules DO through EP. For instance, the GO module starts at $4225, DO begins at $40D6, and EP starts at $43C1.

$4CB6–4E3B — **ASCII PROMPTS AND DOS COMMANDS**

These ASCII messages range from "Insert sheet, press return" through "Proceed / Y = Replace". They are selected as needed to prompt the user or to activate DOS.

$4E3C–4EC9 — **WPL ERROR MESSAGE FILE**

This stash holds all the WPL error messages, starting with a prompt of "WPL Error:" and ending with "Glossary nesting." When an error happens, the prompt is put down, and is then followed by one of the error messages.

$4ECA–4F84 — **DOS FUNCTIONS MENU**

This stash holds all the ASCII characters needed for the [O] DOS access command menu.

$4F85–4FE8 — **VARIOUS ASCII STASHES**

Holds "loose end" ASCII prompts and messages including the help DOS filename, the STARTUP file name, and two user prompts.

$4FE9–5087 — **FIRST SCREEN**

Contains an ASCII image of the first screen displayed on a cold start.

$5088–508B — **APPARENTLY UNUSED**

These four $00 bytes are apparently unused in this particular version.

$508C–51F9 — **ADDITIONAL FUNCTIONS MENU**

Contains an ASCII image of the additional functions menu and selection prompt.

$51FA–520F — **ADDITIONAL FUNCTION ADDRESSES**

Holds the address pairs needed to enter each additional function routine, ranging from $2EC1 to load the tab file, through $2CA4 to quit.

$5210–5211 — **EXPRESS CURSOR MOTION WORK FILE**

This misplaced work file holds two stashes used by the express cursor motions. $5210 is a line counter, usually set to 12 lines. $5211 is an abort file, holding an $A0 for stop on space, or $00 for stop on file end.

Table 12-3—cont. **Apple Writer IIe Reference File Details**

$5212–5213	**— PRINTING WORK FILE**
	This misplaced work file holds two stashes used by the [P]rint routines. $5212 holds a copy of the left margin LM value, while $5213 keeps the right margin RM value. These two are subtracted to find the default line length for the top and bottom header margins.
$5214–53CF	**— PRINT/PROGRAM FUNCTIONS MENU**
	Contains the ASCII image of the print/program functions screen. Values are added to this background display during the PP "?" command.

screen and its address in memory. Any program has a choice of calculating the leftmost screen line base address, or else looking that value up in a table. Table lookup is much faster. The process is called BASHing. The BASH address for the top line is $0400. The next one down is $0480, and so on down the list.

The next reference file is both a menu and a prompting list of available functions. There are 32 possible control command functions. Those used range from [@], which is really the delete key, on up through [Z], the wraparound toggle.

Some of the control commands are missing and some are "hidden" as dedicated keys. Here is a summary of the . . .

"Funny" Control Commands

Dedicated use—

[@] — is the delete key
[H] — is the left arrow
[I] — is the tab key
[J] — is the down arrow
[K] — is the up arrow
[U] — is the right arrow

Not available for use —

[A] — is saved for a modem
[M] — is a carriage return

The DELETE key is recoded as a $80 because its $FF value is reserved as a text file marker. The arrow keys are really the control functions [H], [J], [K], and [U] as shown. [I] is the tab key that does the actual tabs.

We will be using the WPL way of showing control commands in this enhancement. Thus, "[T]" means "press and hold down the control key. Then press and release the T key. Then release the control key."

[A] is not used at all, since many modems need this command for telecommunications. And, finally, [M] is really a carriage return. Since a carriage return signifies the *end* of any user response, it is not normally available for anything else. If you must imbed an unusual carriage return into your text, you can do so with the "]" in the glossary. You could also use one of the special delimiters available for the [F] replace option.

The function list is used two ways. It is first scanned to try and find a match on a control key. If a match is found, the action is done. If no match is found, any action is

ignored. Secondly, some functions need a user prompt, such as when "[S]ave:" asks you for a filename. These prompts are placed on screen when and as needed.

A list of function addresses follows the function list. These are detailed in Table 12-6. When a match is found between a control key and the function list, the function address is jumped to. This carries out the requested action.

Next in the reference file is a pair of back-to-back matching files. The first of these holds two letter pairs of print constants, while the second holds two letter WPL commands. On a print constants match, the value is taken, converted to hexadecimal, and stored as needed in the print constants work file. On a WPL command, the needed action is carried out.

A list of WPL addresses follows the WPL command list. As before, on a two-letter match, that action is done. Table 12-6 shows you which action goes where.

The reference file continues with some ASCII coded prompts and DOS commands, followed by the WPL error messages. These characters are grabbed when needed, either for a user prompt, or to issue a DOS command. The DOS function menu needed by [O] follows, along with some more ASCII stashes.

All of which brings us up to the startup screen at $4FE9. This holds the first screen image you see on bootup. Four unused bytes follow the image.

The Additional Functions menu needed by [Q] is next, followed by the address list of each additional function routine. These are followed by a short pair of misplaced work files.

Ending the reference files, the Print/Program menu starts at $5214. This image is used to put down the fixed portions of the Print/Program menu. The changing portions are read as needed from the Print/Program work file.

As we have just seen, the reference files hold things that seldom if ever change during program use. The reference files are loaded into the machine, following the main word processing code.

Note how you can use your cross-reference and the avalanche effect on these work files. Knowing what each file is used for lets the cross-reference list nail down which module uses which reference file for what purpose.

Remember that the reference files and internal files are loaded off disk, while most of the work files are created, initialized, and then used by the running program.

That just about completes our look into the various file areas. Reviewing, there is a text file area that holds the actual words being processed. There is a work file area that keeps things that are likely to change. There is the internal file area, and finally, there is the reference file area that holds things that are rarely changed during the program use.

The next big question is . . .

HOW DO YOU CRACK PAGE ZERO?

Many of the great mysteries to be solved by the tearing method involve strange and wondrous uses of page zero.

Page zero addresses on any 6502 system are very handy, since they are easy to access and use. More importantly, certain address pairs that let you do 16-bit "anywhere in memory" access positively *must* sit on the zero page.

In short, there is no way to understand *any* Apple program if you do not thoroughly understand what is on page zero, how it gets there, and how it is used.

Important uses of page zero include pointers, counters, stashes, and flags . . .

Important Uses of Page Zero
Pointers —
A pointer holds an address or an address pair that is used to find some other memory location.
Counters —
A counter is used to remember positions or trips needed. Very often, these start at some value and are decremented down to zero.
Stashes —
A stash is used to temporarily hold some value that is likely to be changed.
Flags —
Flags remember conditions and operating modes. A flag often has only two or three possible values.

Pointers are used to hold addresses. A single pointer can only address 256 different places in memory. A dual pointer can address 65,536 places in memory. Dual pointers are most often used with the 6502's very powerful indirect indexed addressing mode.

Indirect indexed addressing lets the computer reach anywhere in memory without worrying about page breaks or 256 byte limits. For instance, in Apple Writer IIe, there is a LOCURS pointer that points to the current cursor position in the LOFILE half of the text file.

By the way, if you are rusty on addressing modes and machine code in general, check into *Don Lancaster's Micro Cookbooks* (Sams Cat. Nos. 21828 and 21829). Volume 2 on machine language programming should be of special interest to you.

Counters are used to hold something that is being incremented or decremented until some magic value occurs. Most often, a counter is initialized to its maximum value and then decremented to zero. This is done since zero is testable "free" with the BNE command. In Apple Writer IIe, there is a deletion counter that makes sure you do not delete words or paragraphs that are longer than 1024 characters.

Stashes hold values that may or may not change. The advantage of keeping values on page zero, rather than elsewhere in memory, is that you can reach these values faster with fewer bytes of code. In Apple Writer IIe, there is a read-only stash that remembers whether you have a 40 or an 80 column screen width.

Flags remember conditions for you. Usually, a flag will only have a few possible values, with a "don't" value of $00 and a "do" value of $FF common. In Apple Writer IIe, there is a "R" flag that remembers whether or not you are in the replace mode.

The big question is "How do you find out what the flags do?"

This is one route to . . .

Cracking Page Zero Locations

(1) Create a notebook of all used locations as listed on the cross-reference for the study program.

(2) Do the tearing method, putting all known information about all flag uses into the notebook.

(3) Go back to the cross-reference and color code each page zero use, red for writes and green for reads. Correct any errors or omissions in the notebook.

(4) Write a complete script of page zero use. Use both a summary list and a detailed script.

As with most other tearing method actions, it takes a few passes to get it right. The first pass finds out if something is used. The second pass finds out roughly who uses that location for what. The final pass nails down the exact use details.

I like to take a small old notebook, and put two or three page zero addresses on each sheet. You get these uses from your page zero cross reference.

As you go through the tearing method, some page zero uses will immediately leap out at you. For instance, in Apple Writer IIe, the [V] command vectors to $3274 which changes the state of the $D0 flag from $00 to $FF or vice versa. Obviously $D0 is our [V]erbatim flag, and we are home free on this one.

Record everything of interest you find in the notebook. Even if you don't have the foggiest what the location is up to, knowing that [L]oad and [S]ave need it and that it gets inited to some value will help bunches later.

The notebook should completely crack about one-third of the page zero locations. Another third should be "pretty nigh, but not plumb." And the final third will still have some mystery surrounding them. Regardless of how far you get, complete the main tearing process as far as you possibly can.

Pay particular attention to whether a page zero location is global or local . . .

Global Value —

Something that has only one possible meaning or use during the entire program.

Local Value —

Something that can have many different meanings at different times in different points in the program.

Global page zero locations have only one single use or meaning. Local page zero locations can have different uses *at different times*, when referenced by different places in the program. Globals are obviously simpler to handle. If you find the same location being used by two or more wildly different portions of the code, assume it is a local value. Then prove yourself right or wrong.

As Apple Writer IIe use examples, the [R] flag at $F5 is used everywhere in the program to pick inserting versus replacing. The Y-register stash at $C5 is used many different places in the program for many different things.

Note that the second page zero location in a pointer pair often goes along free for the ride. Thus, you might initialize both $80 low and $81 high as a pointer pair, but you would only refer to the $80 in a LDA (80),Y command. The use of $81 as a page location or "high address" is inferred in the LDA (80),Y command. This is true for most double wide pointer uses.

O.K.

At this point, your page zero should be around half cracked. Less than a third white margin should remain in your main tearing attack.

Next, go back to your cross-reference listing, and start color coding each page zero use. Do this one location at a time. Red for writes, green for reads. As you color each location, update the notebook with who uses what how.

Pay particular attention to what *sets up* the location, who *changes* the location, and which code *tests* that location. This should make the use of each flag obvious. Correct and expand your notebook comments as you crack each location.

Remember that any booting or setup code can init certain locations to certain values. Most of the Apple Writer IIe page zero locations above $80 init to $00. If you have a "read-only" or "all green" location, always go back to the booting and setup code to see what got stashed where. If some location seems always to be "stuck" in a certain value, make sure there is no branching or indirect stores that change it in a subtle way. Also, find out if different or older versions of the program might have needed this location for obsolete uses.

We won't be looking at the booting code much here, because of the abuse potential of this information. If you go through the tearing process for this program, finding out how the boot works ends up both trivial and simple.

A joke, even.

At any rate, the way you crack page zero is to first record the obvious and make some guesses, and then go back and study each page zero location one on one.

Let's do it . . .

APPLE WRITER IIe PAGE ZERO USES

Table 12-4 gives you a summary of the page zero uses of Apple Writer IIe, broken down into pointers, counters, stashes, and flags. Each listing is then shown in order of increasing address.

More detail appears in Table 12-5, which gives you a complete script of all the page zero uses.

I guess a program counter is really a pointer, so there is some overlap between the pointers and the counters. And the line between a multivalue flag and a few-valued stash also gets a mite thin at times.

Whatever.

Probably the single most important page zero locations are $84 and $85, the LOCURS pointer. This pointer pair points to the address in the text file in auxiliary memory where the next character is to be entered or removed.

As is customary in the 6502 world, the "low," or "position" address appears first in $80 and the "high" or "page" address is stashed in $81. For a use example, if $80 holds a $46, and $81 holds a $35, and the Y register holds an $00, the command LDA (80),Y goes to address $3546 and puts a copy of what it finds in the accumulator. Should the Y register have been holding an $02 instead, the same command would go

Table 12-4. Apple Writer IIe Page Zero Use Summary

A summary of page zero use follows, with much more detail on each location provided in Table 12-5, 12-7, and in your own cross reference listings. Locations marked with an "*" have more than one use and should be approached with caution.

Here goes:

	— pointers —
*$00,01	— General use local pointer
$05	— WPL label pointer
$26,27	— Screen scrolling pointer
$28,29	— Screen base address pointer
$36,37	— COUT monitor destination
$38,39	— KEYIN monitor source
*$80,81	— General use local pointer
*$82,83	— General use local pointer
$84,85	— LOCURS text file pointer
$86,87	— HICURS text file pointer
$88,89	— Screen text file pointer
$8E	— String $A-$D pointer
$90,91	— Printer text file pointer
$92	— WPL subroutine stack pointer
$94,95	— Word deletion buffer pointer
$9E,9F	— Print destination pointer
$A2	— First delimiter pointer
$A3	— Second delimiter pointer
$A4	— Third delimiter pointer
$A6,A7	— Text file HIMEM pointer
$AA,AB	— User prompting pointer
$AC	— Character deletion pointer
$B1,B2	— Text file LOMEM pointer
$BA,BB	— DOS utility pointer
$E9	— Last character pointer
$F0	— DOS pointer to loaded file
$F1	— DOS pointer to T/S file
$F2	— Type-ahead emptier pointer
$F3	— Type-ahead filler pointer
*$F9,FA	— General use local pointer
	— counters —
$06	— TL,BL character counter
$96,97	— Tab position counter
$A0–A1	— WPL Program counter
$BE,BF	— Page number counter
$C8	— Screen line counter
$DA	— Printed line counter
$EF,F0	— Deletion overload counter
	— stashes —
*$00	— Multiuse local stash
*$01	— Multiuse local stash
*$02	— Multiuse local stash
$22	— Screen window top stash
$23	— Screen window bottom stash
$24	— Cursor horizontal position
$25	— Cursor vertical position
*$34	— Y-Save stash
*$35	— X-Save stash
*$80	— Disk volume save
*$82	— Multiuse local stash

Table 12-4—cont. Apple Writer IIe Page Zero Use Summary

$8A	— Screen HPOSN save
$8B	— Screen VPOSN save
$8C	— WPL current character save
*$98,99	— End address stash
$9A,9B	— Memory left stash
$B5	— Cursor symbol stash
$B6	— Screen width stash
$B7	— Left margin padding stash
$B9	— Screen cursor position save
$C0,C1	— Hexadecimal stash
*$C2	— General use local stash
*$C5	— Y-Register stash
*$C6	— X-Register stash
*$C7	— Accumulator stash
*$D3	— General use local stash
$DB	— Last printed line stash
$DC	— Line length stash
$E6	— Delimiter stash
$EA	— Wildcard stash
$EB	— Fake carriage return stash
$EC	— Any length stash
$F4	— Busy prompt symbol stash
$FB	— Open/Solid Apple save
	— flags —
*$03	— Local use flag
$7F	— DOS clone flag (see text)
*$80	— Catalog to text file flag
$A8	— Arithmetic mode flag
$AD	— String source flag
$B3	— System configuration flag
$B4	— DOS version flag
$B8	— Printer enable flag
$C4	— Case flag
$C9	— Any length work flag
$CE	— Screen source flag
$CF	— Data direction flag
$D0	— Verbatim flag
$D2	— Carriage return display flag
$D4	— Cease printing flag
$D5	— Load from memory flag
$D7	— String $A–$D load flag
$DD	— First paragraph line flag
$DF	— WPL/glossary active flag
$E0	— Underline flag
$E1	— Wraparound flag
*$E2	— Append or All flag
$E5	— Data line display flag
$E7	— WPL continue flag
$E8	— Case change flag
$ED	— Mystery flag
$EE	— Deletion overload flag
$F5	— Replace mode flag
$F6	— String $A-$D active flag
$F7	— Screen display flag

Table 12-4—cont. Apple Writer IIe Page Zero Use Summary

$F8	— Split screen flag
$FD	— Screen destination flag
$FE	— Footnote buffer flag
$FF	— Bottom of page flag

Table 12-5. Detailed Script of Apple Writer IIe Page Zero Use

Page zero is used for pointers that hold address values, counters that keep track of positions, stashes that hold constants or characters, and flags that remember conditions and modes. Single page zero locations are used for eight or fewer bits of information. Double page zero locations are used to hold 9 to 16 bits of information.

Most of these locations above $80 are inited to $00 during the boot process. A $00 on a flag usually means "don't."

Here is a rundown:

$00 — **MULTIUSE LOCAL STASH**

Local stash when used by itself. Line counter for glossary display. [S]ave character stash. Character stash for OPEN and APPEND. Force case inhibitor. [L]oad character stash. Header space formatting. PP value hold. IN string address calculator. AS string length. Position counter for center and right of TL or BL. Menu selection save for DOS [O].

$01 — **MULTIUSE LOCAL STASH**

Local stash when used by itself. Row counter on title border. Tab position compare. Status line position counter.

$00–01 — **MULTIUSE LOCAL POINTER**

A 16-bit wide pointer when used as a pair. [Q] pointer prompt. Memory destruct pointer. DOS file manager pointer. WPL or glossary disk load destination address. Footnote buffer pointer.

$02 — **MULTIUSE LOCAL STASH**

Holds the quit charater on [W] and [X]. A slot and drive prompt flag for [O]. Tab position for [I]. Keybuffer pointer in [F]. WPL string offset address stash.

$03 — **DUAL USE TAB/STRING FLAG**

On an [I] tab, a $00 value means no tab value available. A $FF means tab can be completed. On the IN and AS commands, the $FF value means to read the string from the WPL file. A $00 value means normal string access from the keyboard buffer.

$05 — **WPL LABEL POINTER**

Points to the first character in the label of a WPL command. Used by SR subroutine search and for the LABEL NOT FOUND error message.

$06 — **TL/BL CHARACTER COUNTER**

Used in formatting the top and bottom line messages for left, center, and right positions.

$0D — **INCLUDE DELIMITER FLAG**

On a [L]oad, a $00 value here means to load including delimiters. A $FF value means to omit delimiters from the load.

$22 — **SCREEN WINDOW TOP STASH**

Holds the top of the screen window. Inits to $00 and is modified by [Y] for split screen display.

$23 — **SCREEN WINDOW BOTTOM STASH**

Used as a compare value to stop screen entry. An $18 for full screen or bottom of split screen. An $18 or a $0C for split screen, corresponding to 12 or 24 display lines.

$24 — **CURSOR HORIZONTAL POSITION STASH**

Holds the current screen horizontal position, ranging from $00 extreme left through $4F for 80-character extreme right. Altered by nearly all screen entries and cursor motions.

$25 — **CURSOR VERTICAL POSITION STASH**

Holds the current screen vertical position, ranging from $00 at top through $17 at full screen bottom. Altered by many screen entries and cursor motions.

Table 12-5—cont. **Detailed Script of Apple Writer IIe Page Zero Use**

$26,27 — **SCREEN SCROLL POINTER**

Holds the destination address during screen scrolling. Inits to present line position. Line below is mapped up one, repeating as needed to complete scrolling.

$28,29 — **SCREEN BASE ADDRESS POINTER**

Holds the memory address of the leftmost position on the current screen line. Found by table lookup from the BASH table at $4AF8–4B28. Used to enter characters, read from screen, flash cursor, and in screen scrolling.

$34 — **Y-SAVE STASH**

Used to save the contents of the Y register during access to the type-ahead buffer, KSWL entry, and character-to-screen entry.

$35 — **X-SAVE STASH**

Used to save the contents of the X register during access to the type-ahead buffer, KSWL entry, and character-to-screen entry.

$36–37 — **COUT MONITOR DESTINATION POINTER**

Decides where characters are to go when using the monitor COUT routine at $FDED. Normally points to an immediate RTS at $FDFF for access to DOS only. Changes to $260B for display to screen, to $274B for a CATALOG display, and to $C100 for a printer in slot 1 for the keyboard direct to printer option.

$38–39 — **KEYIN MONITOR SOURCE POINTER**

Inits to $2565, the KSWL routine in Apple Writer IIe. Not otherwise referenced, since this program uses its own internal KEYIN routine with an internal type-ahead buffer.

$7F — **DOS CLONE FLAG**

The intended use of this flag was to pick which DOS image to use. A $00 value uses the DOS image in main memory for most DOS access. A $FF value was supposed to use the DOS image clone in auxiliary memory for a text file [S]ave.

Actually, there is no auxiliary DOS image, so this flag does nothing at all. See text.

$80 — **DUAL USE STASH or FLAG**

When used by itself, is a catalog flag, with $00 being a normal catalog, while $FF also catalogs to the text file. Separately, this location holds a disk volume number during [L].

$80–81 — **GENERAL USE POINTER**

Used any time a temporary 16-bit pointer is locally needed. Inits to $1B00 for glossary access. Does the forced jump in [Q]. Clones memory for 1.1 updates. Handles <ctrl> matches and screen prompts. Tab file limit check. Cursor eraser to force high ASCII. File to screen pointer. Load file pointer. PP menu pointer. IOB printer.

$82 — **DUAL USE LOCAL STASH**

When used by itself, locally holds the DOS error message number. Also used as a volume hold and compare in the volume verify routine.

$82,83 — **GENERAL USE POINTER**

Used as a local pointer. Picks the <ctrl> or PP address selected as a forced jump. A pointer for DOS access of the text file.

$84,85 — **LOCURS TEXT FILE POINTER**

Points to the top of LOFILE, which is the currently cursed character. Used for every major access to LOFILE, all entries, to the text file, and all cursor motions.

$86,87 — **HICURS TEXT FILE POINTER**

Points to the bottom of HIFILE, which is one past the currently cursed character. Used for insertions, deletions, and any other time the cursed location is not at the end of the text file.

$88,89 — **SCREEN TEXT FILE POINTER**

Points to the location in the text file about to be put on the screen. Inits to 6 or 12 lines before LOCURS, depending on screen split. Keys on low-ASCII markers at the end of each screen line in the LOCURS and HICURS text files. Maps LOFILE up to the center of the screen and the cursed location. Then switches to HIFILE and maps the rest of the screen.

$8A — **SCREEN HORIZONTAL POSITION STASH**

Holds save of cursor horizontal position during screen update.

$8B — **SCREEN VERTICAL POSITION STASH**

Holds save of cursor vertical position during screen update.

Table 12-5—cont. **Detailed Script of Apple Writer IIe Page Zero Use**

$8C	—	**WPL CURRENT CHARACTER STASH** Remembers the current WPL character being evaluated. Used to search for a "=$" string assignment and to end on a carriage return.
$8E	—	**STRING $A–$D POINTER** Points first to the starting point of each string in the $A–$D buffer work file $1780–187F. Then acts as a character pointer as that string is used. Inits to $00 for a string $A, $40 for string $B, $80 for string $C and $C0 for $D.
$90,91	—	**PRINTER POINTER TO TEXT FILE** Points to the character to be printed in the text file. Inits to LOFILE start at $0801.
$92	—	**WPL SUBROUTINE STACK POINTER** Points to address pairs in the WPL stack work file at $1700–1740. Zeros on init. A 6 bit pointer, limited to 64 values, and pointing to 32 possible pairs of WPL return addresses.
$94,95	—	**WORD DELETION POINTER** This round-and-round pointer always stays in the word deletion work file $0800–0BFF. Advances each time a character is added to the deletion buffer and backs up each time one is removed.
$96,97	—	**TAB COUNTER** Holds the character position since the last carriage return. Used by the tab routines and by the tab display. Updated on each character entry or deletion. Note that a pair of counter values is used so you can tab past 255 into a long paragraph or complicated form.
$98,99	—	**ENDPOINT ADDRESS STASH** Used to hold the endpoint address needed by [F], [S], and [Y]. Inits to the $F9 pointer for split screen use, and to the present cursor position for find and save.
$9A,9B	—	**MEMORY LEFT STASH** Subtracts HICURS-LOCURS to find out how much memory is left. Used by the status line.
$9E,9F	—	**PRINT DESTINATION POINTER** Holds the starting address of a printer routine. Normally $C100 for most printers, $FDED for the print-to-disk option, and $269F for the print-to-screen option. Set by PD.
$A0–A1	—	**WPL PROGRAM COUNTER** Points to the next character in the WPL work file starting at $0E00. Inits to $0E00 and is saved to the WPL stack on a subroutine call. Also is restored from the WPL stack on a sub return. Changed on the GO command.
$A2	—	**FIRST DELIMITER POINTER** Used to find the first delimiter and to scan between the first and second delimiters in the keybuffer during [F]ind and [L]oad. Also used locally as a PP first letter command stash.
$A3	—	**SECOND DELIMITER POINTER** Used to find the second delimiter and to scan between the second and third delimiters in the keybuffer during [F]ind and [L]oad. Also used locally as a PP second letter command stash.
$A4	—	**THIRD DELIMITER POINTER** Used to find and hold the third delimiter during [L]oad.
$A6,A7	—	**HIMEM POINTER** Inited to $BEFF by the boot process to set the HIFILE upper limit. This location is read only.
$A8	—	**ARITHMETIC MODE FLAG** Remembers whether a numeric value is absolute or relative. A $00 means an absolute value. A $AB for "+" or a $AD for "−" handle relative values. Note that either of these sets the N flag. Negative values are automatically twos complemented during entry so that a simple add-only routine can handle both "+" and "−".
$AA,AB	—	**PROMPT POINTER** Points to the text prompts needed by the bottom window on [ ] commands.
$AC	—	**SWALLOW BUFFER POINTER** Points to last used location in the single character swallow buffer at $0300–$037F. MSB is ignored, forcing pointer round and round in the buffer. This pointer gets incremented on single-character deletions and gets decremented on single-character insertions.

Table 12-5—cont. Detailed Script of Apple Writer IIe Page Zero Use

$AD	— **STRING SOURCE FLAG**	A $00 value gets a new string from the user. A $FF value uses the old string already sitting in the $0200 keybuffer.
$B1,B2	— **LOMEM POINTER**	Boot process inits this address pair to $0800, the start of LOFILE in auxiliary memory. These values are read only.
$B3	— **SYSTEM CONFIGURATION FLAG**	Boot process inits this flag to $C0 for a 128K, 80-column Apple IIe. Used in other versions for screen formatting and memory management. Read only.
$B4	— **DOS VERSION FLAG**	Stays at $40 under DOS 3.3e. In other versions, a cleared V flag will run the ProDOS program named STARTUP. Also used by DOS error processor to properly link the chosen DOS in use.
$B5	— **CURSOR SYMBOL STASH**	A $20 value here means to use a software flashable white box cursor. A value of $00 means to not alter or change the cursed location.
$B6	— **SCREEN WIDTH STASH**	Inits by boot process to $50 for an 80-character wide screen. Other versions use a $28 value for 40 columns. This location is read only.
$B7	— **LEFT MARGIN PADDING STASH**	Holds the number of print spaces needed at the left margin. Inits to LM. Modified by PM and CJ.
$B8	— **PRINTER ENABLE FLAG**	A $00 value means the printer is off or that the screen is to be the print destination. A value of $FF means the printer is active and pointed to by $9E,9F.
$B9	— **SCREEN CURSOR POSITION STASH**	Holds the point in the cursed line where a switch from LOFILE to HIFILE is needed.
$BA,BB	— **LOAD POINTER or DOS POINTER**	Dual-use pointer pair. Used to move copy of DOS to auxiliary RAM. Separately used as a searching pointer when loading from memory.
$BE,BF	— **PAGE NUMBER COUNTER**	This pair inits to PN and is incremented automatically on each new page printed.
$C0,C1	— **HEXADECIMAL STASH**	Holds a 16-bit hex input for hex-decimal conversion. Receives a 16-bit hex result for decimal-hex conversion. Related decimal values sit in $17E0–17E4.
$C2	— **KEYSAVE or LOCAL STASH**	General use local stash used as a command or character local hold, sort of as an auxiliary accumulator.
$C4	— **CASE FLAG**	Picks case in use. $00 is normal mixed upper and lower. $80 is uppercase only. $C0 is lowercase only.
$C5	— **Y-REGISTER STASH**	General use local stash, most often used to preserve old contents of Y-register. Also a width-of screeen counter used during screen wraparound formatting.
$C6	— **X-REGISTER STASH**	General use local stash, most often used to preserve old contents of X-register. Also used to hold a previous character during screen wraparound formatting.
$C7	— **ACCUMULATOR STASH**	General use local stash, most often used to preserve old contents of accumulator. Also used as a padding flag in FJ.
$C8	— **SCREEN LINE COUNTER**	Inits to $18 lines for full display, or $0C lines for split. Loses one count if data display is on. Decremented to zero as lines are added to the screen.

Table 12-5—cont. **Detailed Script of Apple Writer IIe Page Zero Use**

$C9	— **ANY LENGTH WORK FLAG** An $FF value means that an any length search is in progress. A $00 value stops the any length search.
$CE	— **SCREEN SOURCE FLAG** A $00 value uses the LOFILE to get screen characters up to and including the cursed position. A $FF value uses the HIFILE to get screen characters from just past the cursor to the end of the screen.
$CF	— **DATA DIRECTION FLAG** A $FF value sets the arrow to the right for forward searches and to retrieve text from the word and paragraph buffer. A $00 value sets the arrow to the left for backwards searches and word or paragraph text deletion.
$D0	— **VERBATIM DISPLAY FLAG** A $00 here allows normal use of control characters as commands. A $FF puts any control characters directly into the text, exactly as entered. Used to imbed special printer and typesetting commands.
$D2	— **CARRIAGE RETURN DISPLAY FLAG** A $00 value here displays text normally on the screen, without showing carriage returns. A $FF here shows all carriage returns as "]" symbol. Used to resolve wraparound and tabbing problems.
$D3	— **DUAL-USE STASH** This byte has two separate uses. It is a string length counter for [F]ind, and a line counter used by [P]rint. Each use is local.
$D4	— **CEASE-PRINTING FLAG** A $00 value here allows printing to continue. A value of $FF stops printing, usually because no text is left or the escape key has been hit.
$D5	— **LOAD FROM MEMORY FLAG** A $00 value here does a normal load from disk, while a $FF value loads from the text file. Used to copy text without moving the original.
$D7	— **STRING $A–$D LOAD FLAG** A $00 value here gets string values from the keyboard as usual. The $FF value loads the string from the WPL program file. Used to assign values to the $A through $D strings.
$DA	— **LINE COUNTER** Keeps track of the current vertical position on a page being printed. Inits to $00 and is incremented each line. Compared against the last-line stash to end a page.
$DB	— **LAST-LINE STASH** Holds the bottom printed line position. One line is substracted if BL is in use. Two lines are subtracted for the first footnote and one extra line for each additional footnote. Decides how many text lines go on the page.
$DC	— **LINE LENGTH STASH** Inits to the number of characters per line, by subtracting RM-LM. Used by the line justify routines.
$DD	— **FIRST LINE IN PARAGRAPH FLAG** Inits to $FF at the start of each paragraph. Returns to $00 after the first line. This flag is used to set paragraph margins and to test for imbedded print commands.
$DF	— **WPL or GLOSSARY ACTIVITY FLAG** A $00 value means that neither WPL nor the glossary is active or in use. The $80 value means WPL is active and is usually used to bypass screen prompts. The $40 value means that the glossary is active and that it is to be the source for any needed characters.
$E0	— **UNDERLINE FLAG** A $00 value means no underlining is in use, while a $FF value activates the underline mode. Underlining is done by individually backspacing and underlining each character. The UT value toggles this flag. Present flag value is held during TL, BL, or a footnote.
$E1	— **WRAPAROUND FLAG** A $00 value gives you a normal display of whole words only. A $FF value breaks the words as needed at the right margin.

Table 12-5—cont. Detailed Script of Apple Writer IIe Page Zero Use

$E2	— **DUAL-USE APPEND or ALL FLAG** This flag has two separate uses. On [S]ave a $00 value means to save to a new file, while the $FF value means to append an existing file. On both [L]oad and [F]ind, a $00 value means to handle one occurrence only. The set of $FF value means to handle all occurrences. Both uses are local and independent.
$E5	— **DATA LINE DISPLAY FLAG** This three-state flag decides which data line is to be displayed at the top of the screen. A $00 value displays nothing. A $80 value shows the usual "Mem-Len-Pos File" header, while the $C0 value shows the tab position display.
$E6	— **DELIMITER STASH** Used to hold the delimiter match character. A $00 value means no delimiter. The ASCII value of the delimiter is held when used.
$E7	— **WPL CONTINUE FLAG** This flag is set to zero on the DO and GO commands. It clears WPL if it ever gets into the $FF state. Apparently not used.
$E8	— **CASE CHANGE FLAG** A $FF value here means to change the case per the case flag at $C4. A $00 value means to use normal, mixed upper and lowercases. This seems to be a leftover from an earlier version as this flag is never read.
$E9	— **FILENAME LAST CHARACTER POINTER** Points to the final character in the keybuffer. Used in [S]ave to test for the " + " for append, and in [L] to test for the "/" to do a load using the screen as a source.
$EA	— **WILDCARD STASH** Holds the match character for a legal wildcard "any character" search. Holds a $03 when using the "/" standard delimiter that does not allow wildcards.
$EB	— **FAKE CARRIAGE RETURN STASH** Holds the match character for a legal substitute carriage return. Holds a $02 when using the "/" standard delimiter that does not allow carriage return substitution.
$EC	— **ANY LENGTH STASH** Holds the match character for any length of text screen. Holds a $01 when using the "/" standard delimiter that does not allow any length searching.
$ED	— **MYSTERY FLAG** Zeroed at entry into word processing code. Not otherwise referenced in this version. Was the screen mode case flag in an earlier version that did not have a totally live screen.
$EE	— **DELETION OVERLOAD FLAG** Inits to $00. Goes to $FF if a paragraph or a word deletion > 1024 characters is attempted.
$EF–F0	— **DELETION OVERLOAD COUNTER** These two locations init to 1024 and count down. If a word or a paragraph deletion > 1024 characters is attempted, the deletion overload flag is set. A separate counter is needed since the actual deletion process goes round and round in the deletion buffer. Note that $F0 has a second and separate use as a DOS pointer.
$F0	— **DUAL-USE DELETION COUNTER or DOS POINTER** This location has two separate and local uses. During [W] or [X] paragraph deletions, it forms the high byte of a 1024 counter, working with $EF. During RWTS access, this pointer points to a character in the DOS text file buffer sitting at $0D00–0DFF.
$F1	— **DOS T/S LIST POINTER** Points to address pairs on the DOS track and sector list image at $0C00–0CFF. Inits to $0A and is incremented twice per track and sector entry, starting at $0C. Used to load one page of a DOS text file during [L].
$F2	— **TYPE-AHEAD BUFFER EMPTIER** Points to the last used character in the type-ahead buffer at $1740–177F. Limited to 6 bits for 64 possible round-and-round locations. Gets "behind" $F3 when busy. Increments on each use of a character.
$F3	— **TYPE-AHEAD BUFFER FILLER** Points to the next available character location in the type-ahead buffer. Six bits for 64 round-and-round file values at $1740–177F. Increments on each keystroke that cannot be immediately used.

Table 12-5—cont. Detailed Script of Apple Writer IIe Page Zero Use

$F4	— **BUSY PROMPT** Holds an ASCII space or $A0 when not busy and the type-ahead buffer is not in use. Holds an ASCII "*" or $AA when busy and the type-ahead buffer is being emptied.
$F5	— **REPLACE MODE FLAG** A $00 value here does normal character entry by insertion. A $FF value enters characters by writing over, or replacing, the existing character. Toggled by [R] and reset by just about all cursor commands.
$F6	— **STRING $A–$D ACTIVITY FLAG** Inhibits WPL interpreter when $A–$D strings are being processed. A $00 means normal WPL use; a $FF here means a string is being processed.
$F7	— **SCREEN DISPLAY FLAG** A $00 value here means to not display to screen. $FF means to do the normal screen display. Involved with the YD and ND commands.
$F8	— **SPLIT-SCREEN FLAG** A three-valued flag that sets the screen display mode. A $00 means a normal full screen. A $FF means a split screen with the bottom half active. A $7F value means a split screen with the top half active.
$F9,FA	— **DELETION or SPLIT-SCREEN POINTER** A general use 16-bit address pointer used in several different and local ways. Used as a deletion pointer for character, paragraph, and word removal. Also used as a split-screen pointer to hold the starting address of the inactive half of the split screen.
$FB	— **APPLE KEY STASH** Used to hold open-apple/solid-apple keystrokes when the type-ahead buffer is needed. A $00 means no key depressed. A $40 value means the open-apple key is down. A value of $80 means the closed-apple key is down. Open apple is used to get help, to swallow and retrieve single characters, or to activate the glossary. The closed-apple key is used to tab over text and to copy text, as well as for express cursor movements.
$FD	— **SCREEN-ONLY FLAG** Used to load only to the screen under the [L]oad command with a "/" prefix. A $00 value does a normal load into the text file. A $FF command loads only to the screen.
$FE	— **FOOTNOTE BUFFER FLAG** A $00 value means no footnotes are needed. A value of $FF or lower means the footnote buffer is in use. Footnotes are held in the footnote buffer from the time they appear in the text until the bottom of the current page. This flag is decremented once for each footnote.
$FF	— **BOTTOM OF PAGE FLAG** This flag inits to $00 and goes to a $FF when the body is completely printed or on a form feed. Footnotes and the bottom line can then be entered.

to address $3548. Note that this command can reach any location in the entire 64K address space, just by changing the values in $80, $81, and the Y register.

Other double wide pointers are used to access HIFILE, to print characters in order, to display to the screen, and for various other uses that need to access lots of characters in sequence. Be sure you understand exactly how these work. Understanding LOCURS is first and foremost in this quest.

You should now be halfway into your understanding process of this program. We now know how the text file area works. We have studied uses of the work file area, the internal file area, the reference file area, and found out about the many uses of page zero.

Next on the agenda are the . . .

ENTRY POINTS

Entry points are those locations in the code where you go to do something . . .

> Entry Point —
>
> Some location in a block of code that you go to in order to start something happening.

Depending on what you need to get done, there are several possible entry-point levels . . .

> **Entry Levels**
>
> High Level —
>
> Points entered into the whole code by another system to run, rerun or process errors.
>
> Command Level —
>
> Points entered in response to a main menu selection.
>
> Module Level —
>
> Points entered to handle specific tasks or sub-menu selections.
>
> Service Level —
>
> Important subroutines that do all major housekeeping and handle any often-needed utility functions.

Table 12-6 summarizes the important entry points. A complete and detailed disassembly script appears in Table 12-7, which will tell you more than you could possibly want to know about every module in the working code.

There is no single answer to the obvious question, "How does Apple Writer work?" It all depends on what you think is important and where your interests lie. And any attempt to go through the code in numeric order is pretty much fruitless, because you lose track of who is doing what to whom.

Let's instead see if we can't thread together some of the important working concepts of this program. Our first concern should be . . .

DOS

We might start by grabbing the bull by the horns. The DOS 3.3e used in the "F" version of Apple Writer IIe is modified; it is installed in an unusual place; and it is used four different major ways.

Table 12-6. Apple Writer IIe Important Entry Points

Important "F" version entry points include system-level entry that accesses the entire program; command-level entry that does control commands; WPL module entry that handles individual WPL commands; the auxiliary function access; and finally the often-used service subroutines.

Here they are

— system-level entry —

$2300	— Cold-start entry routine
$2303	— Warm-start entry routine
$2306	— DOS error recovery reentry

— command-level entry —

$2AD0	— [@] Unconditional delete
$28DC	— [B] Cursor to start
$3942	— [C] Case changer
$3953	— [D] Data direction change
$2903	— [E] Cursor to end
$3717	— [F] Find, search and replace
$2AE0	— [G] Glossary
$27EC	— [H] Backspace left arrow
$2FAA	— [I] Do actual tab
$280E	— [J] Down arrow
$2816	— [K] Up arrow
$35A0	— [L] Load
$3903	— [N] New
$4A67	— [O] DOS access
$3DC9	— [P] Print/Program main entry
$2C25	— [Q] Auxiliary functions entry
$395A	— [R] Replace mode toggle
$32FC	— [S] Save
$2F2A	— [T] Tab set, clear, or purge
$27F0	— [U] Frontspace, right arrow
$3274	— [V] Verbatim mode toggle
$2A00	— [W] Delete word
$2A00	— [X] Delete paragraph
$31C3	— [Y] Split screen
$3266	— [Z] Wraparound toggle

— WPL modules —

$42DB	— AS Assign string
$41CC	— BL Bottom line
$4414	— CP Continue printing
$40D6	— DO Run WPL program
$43C1	— EP Enable printer
$41FE	— FF Form feed
$4225	— GO Unconditional WPL jump
$429D	— IN User keyboard response
$3F69	— ND Screen display off
$43F7	— NP Begin printing
$41B3	— QT End WPL program
$4274	— PR Prompt to screen
$3F51	— RT WPL subroutine return
$3F06	— SC String compare
$3F6D	— SL String load
$3F33	— SR WPL subroutine jump
$41D0	— TL Top display line
$3F6A	— YD Screen display on

Table 12-6—cont. Apple Writer IIe Important Entry Points

	— auxiliary functions —
$2EC1	— "A" Load tab file
$2EA5	— "B" Save tab file
$2EFB	— "C" Load print file
$2EDF	— "D" Save print file
$2E2A	— "E" Load glossary
$2DB2	— "F" Save glossary
$326D	— "G" Toggle CR display
$325B	— "H" Toggle data display
$2C75	— "I" Keyboard direct to printer
$2D0F	— "J" Convert old 1.1 files
$2CA4	— "K" Quit everything
	— often-used service subroutines —
$2400	— Print character to screen link
$24AC	— Ring the ding dong
$24C0	— Grab I/O hooks
$24E3	— Pick string source
$251A	— Get key from type-ahead buffer
$2565	— KSWL entry point
$260B	— Screen store entry point
$26B2	— BASH calculator from CV
$26B4	— BASH immediate
$26FF	— Init LOFILE
$2725	— Init HIFILE
$2757	— Enter character to LOFILE
$279D	— Move character LOFILE→HIFILE
$27BC	— Zero mark LOFILE end and HIFILE start
$27CA	— Move character LOFILE←HIFILE
$297C	— Increment printer pointer
$2B86	— Turn glossary off
$2C13	— Return prompt
$2E3C	— Get filename, no slot or drive
$2E46	— Get filename, slot, and drive
$2E8E	— Send filename to DOS
$30D8	— Reenter main processor loop
$320B	— Turn split screen off
$32C0	— Prompt screen bottom
$32ED	— Clear bottom of screen
$38E2	— Adjust replace pointer
$392C	— Force uppercase
$3ABB	— Display decimal value
$3ACC	— Store inverse to screen
$3B7E	— Format screen lines
$3B45	— Print WPL error message
$3D93	— Get string for WPL
$400E	— Set print destination to screen
$40B5	— Save old filename to = buffer
$40C2	— Restore old filename
$41B3	— Quit WPL
$421B	— Print space and return
$4783	— Prepare one character for printing
$47E1	— Send character to printer
$47F3	— Close open files
$4951	— Hex-to-decimal conversion
$49FD	— Clear screen
$49FF	— Print normal to screen

Table 12-7. Detailed Script of Apple Writer IIe Main Program

The "F" version of the Apple Writer IIe main program sits between $2300 and $4AF8. The script that follows assumes an 80-column, 128K Apple IIe under relocated DOS 3.3e.

Here is a module-by-module breakdown of the code:

$2300–2302 — **COLD-START ENTRY POINT**

Jumps to actual cold-start code at $2309.

$2303–2305 — **WARM-RESTART ENTRY POINT**

Jumps to actual warm-restart code at $2350. Warm resets reenter here.

$2306–2308 — **DOS ERROR RETURN POINT**

Jumps to DOS error processing code at $480B.

$2309–234F — **COLD-START MODULE**

Init stack pointer. Move memory management code to $0100. Init HICURS and LOCURS cursor pointers. Copy LOCURS to multiple-use pointer. Link DOS vectors and hooks. Verify volume. Clear screen. Grab input and output hooks. Init cursor to flashing box. Put down first screen and prompt for disk change. Load SYS. and TAB. files. Fall through to warm-start module.

$2350–234F — **WARM-RESTART MODULE**

Flip soft switches to write and read RAM. Reset stack pointer. Install memory management code at $0100. Grab input and output hooks. Close open files. Set flashing cursor box. Clear WPL. Close glossary nest. Set screen as printer destination. Jump to main word processing code at $30AB.

$2350–2398 — **INSTALL MEMORY MANAGEMENT CODE**

Move management code block from $2399–23EE in main memory to main page one $0100–0156. Clear the DOS hooks in auxiliary memory, replacing them with a hook to the main DOS error processor. Switch back to writing main memory and return.

$2399–23EE — **MEMORY MANAGEMENT CODE IMAGE**

Gets moved to page one and remains active regardless of whether main or auxiliary RAM is in use. Here are the key access points:

	$0100	— DOS Error recovery
	$0109	— DOS File manager access
*	$0119	— Send character to DOS clone
	$0129	— Read text file byte for screen
	$0132	— Read cursed LOFILE value
	$013B	— Read cursed HIFILE value
	$0144	— Read value for print pointer
	$014D	— Read value for general-use pointer

(* — there is no DOS clone — see text)

$23EF–23FF — **IS TUTORIAL HELP NEEDED?**

Ignore if glossary is active. Test for a "?" or its lowercase "/" equivalent. See if open-apple key is also down. If help is needed, get it via $2423.

$2400–2410 — **PRINT CHARACTER TO SCREEN LINK**

Get character from type-ahead buffer. If it is a $00, ignore and get another one. If WPL is not active, print character to screen.

$2411–2422 — **TRY AND RUN STARTUP**

See if a file named STARTUP is on the diskette. Run it if present. Generate no error message if not, and try only once.

$2423–2441 — **GET HELP**

Zero keysave stash. Force feed "8" to help string for 80-column format. Put "HELP8E,S6,D1" into keybuffer. Call DOS.

$2442–2466 — **PRINT FIRST SCREEN**

Print title border. Home cursor. Print first Apple Writer screen. Change format to suit 80-column display.

$2467–2A46 — **DRAW TITLE BORDER**

Draw a fancy border box on the first screen, changing format to suit 80 columns.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$24A7–24AB	**— PRINT CURSOR BOX TO SCREEN** Get an ASCII $20 and print it to screen. This low ASCII space prints in inverse as an Apple alternate character.
$24AC–24BF	**— DING DONG** Sound the error alarm. Plays two plain, old squarewave bursts back to back, the second lower in pitch. Calls itself for the first note.
$24C0–24E2	**— GRAB I/O HOOKS** Turn the alternate character set (no-flash ASCII) on and test for 128K and 80 columns. Activate same. Connect COUT hooks to a brick wall RTS to output FDFO only to DOS. Connect KSWL,H to point to built-in GETKEY routine.
$24E3–24F4	**— PICK STRING SOURCE** If WPL is active and the $A-$D string load flag is set, get string from WPL file. If not, get string from user at keyboard.
$24F5–2519	**— GET AND SAVE KEY** Save old cursor symbol. Stop cursor flash. Turn on busy flag. Get key via KSWL. Add one count to type-ahead filler. Store character in type-ahead buffer. Store open-apple or closed-apple key in apple buffer.
$251A–2564	**— GET KEY FROM TYPE-AHEAD BUFFER** Save registers. See if buffer is behind. If buffer is behind, add one count to type-ahead emptier; then get the next character from the type-ahead buffer, and fill the apple save flag from the apple buffer. Then restore registers. If buffer is caught up, get the key via KSWL. Turn off the busy flag after checking whether an up-screen or [Y] down-screen status line is active.
$2565–2573	**— KSWL ENTRY POINT** Save the X and Y registers. Get key via internal GETKEY sub. Restore registers. Affirm even character.
$2574–25BC	**— KEYIN KEY GETTER** Check to see if glossary is active. If so, get key from glossary file. Check to see if WPL is active. If so, get key from WPL program file. Otherwise, scan keyboard for a key down, flashing the cursor during the waiting process. Read the open-apple and solid-apple keys, and save them in flag $FB. Read the key and reset the key strobe. If the delete $FF key was pressed, change it to $80. Exit with key value in accumulator.
$25BD–25DB	**— CHANGE CURSOR** Ignore if WPL is active. Divide CH by two to pick even or odd character. If even character, use the main memory. If odd, use auxiliary memory. Read the character, change it from or to inverse, and replace the character on the screen. Note that high ASCII is normal and low ASCII is inverse.
$25DC–25F4	**— PUT CHARACTER DIRECTLY ON SCREEN** Get horizontal position and verify 128K. Divide position by two to pick even or odd character. If even, store in main screen memory. If odd, store in auxiliary screen memory using BASH screen address plus Y register position offset.
$25F5–260A	**— READ CHARACTER DIRECTLY FROM SCREEN** Get horizontal position and verify 128K. Divide position by two to pick even or odd character. If even, load accumulator from main screen memory. If odd, load accumulator from auxiliary screen memory.
$260B–261B	**— SCREEN STORE ENTRY POINT** Save registers. Call screen store logic below. Restore registers.
$261C–266B	**— SCREEN STORE FILTER** If a form feed, clear the screen. If a bell, ring the ding dong. If a carriage return, process unless scrolling is on hold. If a backspace, back up one, fixing the BASH address on underflow. If a printable noncontrol character, put the character on the screen and advance CH. If screen line overflows, do a carriage return.
$266C–267B	**— SCREEN CARRIAGE RETURN** Reset CH and increment CV to get to start of next line. If screen overflows, fall through to scrolling routine below.
$267C–26B1	**— SCROLL SCREEN UP** Calculate screen position and save to screen scroll pointer. Affirm 128K. Move screen up a line at a time, alternating even characters in main memory and odd characters in aux memory, continuing until full screen or [Y] split screen is completely scrolled. Clear bottom line by jumping to the EOL subroutine.
$26B2–26C0	**— BASH CALCULATOR** Convert CV into screen base address by table lookup, saving leftmost screen address to the screen base address pointer.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$26C1–26DF — CLEAR SCREEN WINDOW

Call the clear EOL routine as often as needed to clear whole lines from the present CV position till the lower screen window limit.

$26E0–26EA — HOME SCREEN CURSOR

Move screen cursor to upper left of screen window. Then BASH the CV base address.

$26EB–26FE — CLEAR SCREEN TO END OF LINE

Starting with the present screen position, write $A0 spaces to the end of the current screen line.

$26FF–270D — INIT LOFILE

Set the LOFILE pointer to the first character at $0801. Write a "lower file limit" $FF to $0800 and an "end of LOFILE" $00 to $0801.

$270E–2724 — INIT LOFILE POINTER

Set the LOFILE pointer to LOMEM at $0800 and put a "bottom of file" $FF there. Increment the pointer to point to the first LOFILE character at $0801.

$2725–2733 — INIT HIFILE

Set the HIFILE pointer to the last character at $BEFE. Write an "upper file limit" $FF to $BEFF and a "start of HIFILE" $00 to $BEFE.

$2734–274A — INIT HIFILE POINTER

Set the HIFILE pointer to HIMEM at $BEFF and put a "top of file" $FF there. Decrement the pointer to point to the last possible HIFILE character at $BEFE.

$274B–2756 — CHARACTER TO FILE ACCESS LINK

Save registers. Put character in file with subroutine below. Restore registers. Does not seem to be used in this version.

$2757–279C — PUT CHARACTER IN LOFILE TEXT FILE

Save character. Test for display to screen only. If screen only, do so. If not, test for some remaining memory, sounding alarm and clearing if no room remains. Test for replace mode. If in replace mode, move the character beyond the cursor down to the top of LOFILE. This way, the next character gets overwritten, rather than bumped up. Get the SAVED character back, force high ASCII and make sure it is not the delete key. Then, switch to aux memory and store the character to the top of HIFILE. Increment [W] pointer and LOCURS. Store a $00 to the top of LOFILE.

$279D–27C9 — MOVE CHARACTER FROM LOFILE TO HIFILE

Decrement LOCURS. If LOFILE is empty, then re-init LOFILE and quit. Read character from LOFILE. Test for a case change and do one if needed. Store character to HIFILE in aux memory. Decrement HICURS and store a $00 marker to the bottom of HIFILE.

$27CA–27EB — MOVE CHARACTER FROM HIFILE TO LOFILE

Increment HICURS. If HIFILE is empty, then re-init HIFILE and quit. Read character from HIFILE. Test for a case change and do one if needed. Store character to LOFILE in aux memory. Increment LOCURS and store a $00 marker to the bottom of HIFILE.

$27EC–27FC — TEST BACKSPACE COMMAND

If a backspace command and no apple key, do a backspace once. If a backspace and an open apple, go delete one character. If a backspace and a solid apple, do an express backspace.

$27FD–280D — TEST FRONTSPACE COMMAND

If a frontspace command and no apple key, reset the case flag and do one frontspace. If a frontspace and an open apple, go insert one character. If a frontspace and a solid apple, do an express frontspace.

$280E–2815 — TEST VTAB DOWN COMMAND

If a downtab command and no closed-apple key is pressed, do the downtab once. If the closed-apple key is pressed, try to downtab 12 lines.

$2816–281E — TEST VTAB UP COMMAND

If an uptab command and no closed-apple key is pressed, do the uptab once. If the closed-apple key is pressed, try to uptab 12 lines.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$281F–2840	— **EXPRESS MOTIONS COMMON CODE** This common code is used four ways, force feeding the correct subroutine address for cursor motions in each direction. The open-solid flag is tested. If no solid apple, then each cursor motion is done once. If a solid-apple key is down, then each cursor motion is done many times, with the vertical tabs trying for 12 lines, and the horizontal tabs trying for a full word. Location $5210 holds the trip count while $5211 holds a $00 for “stop at end of file” or a $A0 for “stop at end of word.” Here are the “do it once” subroutines that are called via a force-fed address: $279D — Backspace $27CA — Frontspace $2841 — Tab up $2868 — Tab down Note that the trap at $283E is overwritten by these service subroutine addresses.
$2841–2867	— **MOVE CURSOR UP** Reset the case changer. If wraparound is not in use, transfer one line of characters from LOFILE TO HIFILE. If wraparound is in use, adjust screen line length for correct number of characters to be moved; then do transfer.
$2868–2886	— **MOVE CURSOR DOWN** Reset the case changer. If wraparound is not in use, transfer one line of characters from HIFILE to LOFILE. If wraparound is in use, adjust screen line length for correct number of characters to be moved; then do transfer.
$2887–28A1	— **TRANSFER ONE LINE OF CHARACTERS** Test for forward or backward motion. If forward, try to move 80 characters from HIFILE to LOFILE, aborting on the first carriage return. If backwards, try to move 80 characters from LOFILE to HIFILE, aborting on first carriage return.
$28A2–28BD	— **UPDATE LOCURS POINTER** Init LOFILE pointer. Read LOFILE value. Search up through LOFILE till the $00 end of file marker is reached. Transfer any count residue from Y to the low byte of LOCURS. When finished, LOCURS will point to the present cursed position.
$28BE–28DB	— **UPDATE HICURS POINTER** Init HIFILE pointer. Read HIFILE value. Search down through HIFILE till the $00 start-of-HIFILE marker is reached. Transfer any count residue from Y to the low byte of HICURS. When finished, HICURS will point to the present start of the HICURS file, one character beyond the cursed character.
$28DC–28FF	— **MOVE CURSOR TO BEGINNING OF TEXT FILE** Set the data direction $CF flag to “>”. Decrement LOCURS. Read the LOFILE value and move it to HIFILE. Continue this until you get to the beginning of LOFILE with its $FF marker. Then update both LOCURS and HICURS to their current positions. When finished, all of the characters will be in HIFILE, and LOFILE will be empty.
$2900–2922	— **MOVE CURSOR TO END OF TEXT FILE** Set the data direction $CF flag to “<”. Increment HICURS. Read the HIFILE value and move it to LOFILE. Continue this until you get to the end of HIFILE with its $FF marker. Then update both LOCURS and HICURS to their current positions. When finished, all of the characters will be in LOFILE, and HIFILE will be empty.
$2923–2937	— **COMMON ENDING CODE FOR [B] AND [E]** Add the residue from the Y register to both HICURS and LOCURS pointers. Then put a $00 marker at the top of LOFILE and the bottom of HIFILE.
$2938–2940	— **INCREMENT WORD DELETION POINTERS** Add one to the word deletion pointer, and subtract one from the deletion overload counter.
$2941–2947	— **INCREMENT WPL PROGRAM COUNTER** Adds one to the WPL program counter pair $A0–A1.
$2948–2953	— **TEST SPLIT-SCREEN POINTER** Tests the split-screen pointer and sets the carry flag if this pointer > LOCURS.
$2954–295A	— **INCREMENT LOCURS POINTER** Increments the low cursor pointer by adding one to $84 and, if an overflow, adding one to $85.
$295B–2966	— **INCREMENT SPLIT-SCREEN POINTER** Increment the split-screen pointer only if it is less than LOCURS.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$2967–296D	— **INCREMENT HICURS POINTER** Increments the high cursor pointer by adding one to $86 and, if an overflow, adding one to $87.
$296E–2974	— **INCREMENT SCREEN CURSOR POINTER** Increments the screen cursor pointer by adding one to $88 and, if an overflow, adding one to $89.
$2975–297B	— **INCREMENT LOCAL USE POINTER** Adds one to $80 and, on overflow, adds one to $81.
$297C–2982	— **INCREMENT PRINTER POINTER** Adds one to $90 and, on overflow, adds one to $91.
$2983–299D	— **DECREMENT WORD DELETION POINTERS** Subtract one from the word deletion pointer, and subtract one from the deletion overload counter. If the word deletion pointer underflows, force it to stay on pages $08–0B.
$299E–29A8	— **DECREMENT LOCURS POINTER** Decrements the low cursor pointer by subtracting one from $84 and, if an underflow, subtracting one from $85.
$29A9–29B8	— **DECREMENT SPLIT-SCREEN POINTER** Decrement the split-screen cursor only if it is less than LOCURS.
$29B9–29C3	— **DECREMENT HICURS POINTER** Decrements the high cursor pointer by subtracting one from $86 and, if an underflow, subtracting one from $87.
$29C4–29CE	— **DECREMENT SCREEN CURSOR POINTER** Decrements the screen cursor pointer by subtracting one from $88 and, if an underflow, subtracting one from $89.
$29CF–29D9	— **DECREMENT LOCAL USE POINTER** Subtracts one from $80 and, on underflow, also subtracts one from $81.
$29DA–29FF	— **INIT SCREEN START POINTER** Tries to initialize the screen start pointer to 12 lines behind the current cursor position on full screen, and 6 lines behind on split screen. Stops at the beginning of LOFILE if too near the start.
$2A00–2A0C	— **FIX ERROR BUFFER POINTER** The word and paragraph error buffer must always point somewhere on pages $08-0B. On an out-of-range underflow or overflow, this module forces the pointer round and round.
$2A0D–2A5C	— **DELETE WORD OR PARAGRAPH** Set the deletion overload counter to 1024. Pick an endpoint character of $A0 space for word deletion or an $8D carriage return for paragraph deletion. Test the data direction flag for insertion or deletion. If an insertion, get the characters and enter them. If a deletion, save the characters, deleting them from LOFILE only if not in copy mode. Continue entering or removing characters in a loop until the $A0 or $8D match character, an end-of-previous-line $8D carriage return, a start-of-LOFILE $FF, or 1024 characters. If >1024 characters, ring the ding dong. When done, update the screen.
$2A5D–2A64	— **READ CHARACTER FROM WORD/PARAGRAPH BUFFER** Get a character from the buffer, increment the buffer pointer, and put that character into LOFILE.
$2A65–2A7C	— **COPY MODE FILTER** Test to see if closed-apple key is down. If so, load the word/paragraph buffer without deleting the existing text. Abort on the $FF start of LOFILE marker. If not copy mode, save character to buffer and delete the character from its present LOFILE position.
$2A7D–2A90	— **SAVE CHARACTER TO WORD/PARAGRAPH BUFFER** Back LOCURS up one character. Read that character. Back up word deletion buffer pointer one slot. Save read character to the word deletion buffer. Abort if start of LOFILE. Restore cursors.
$2A91–2AA0	— **RESTORE CHARACTER FROM SWALLOW BUFFER** Decrement the single character deletion pointer, forcing it to $7F on $00 underflow. Get character from swallow buffer and store it to LOFILE. Then update the screen.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$2AA1–2AC3 — SAVE CHARACTER TO SWALLOW BUFFER

Decrement LOCURS and the split-screen cursor. Get cursed character. Force single-character deletion pointer to $00–7F range. Save character to swallow buffer at $0300–03FF. Increment swallow pointer. Abort if at start of LOFILE. Then update the screen.

$2AC4–2ACF — UNDERFLOW ADJUST

Common code area when deletion attempted beyond start of LOFILE. Init LOFILE. Advance split-screen pointer. Update screen.

$2AD0–2ADF — DELETE CHARACTER FOREVER

If not at start of screen, replace the cursed LOFILE character with a $00 and decrement LOCURS. Then update the screen.

$2AE0–2AFD — PROCESS GLOSSARY COMMAND

Clear and prompt the screen bottom. Get user command. Abort on $8D carriage return. If a $BF question mark, go and define new entry. If a $AA purge star, store $00 to start of glossary file at $1B00. If any other character, go and read glossary.

$2AFE–2B2D — FIND GLOSSARY MATCH

Save the glossary match character, aborting on carriage return. Set local pointer to $1B00, the start of the glossary file. Go through the glossary, looking for each character just past a carriage return. Compare that character against the match character. If a match is found, fall through to the glossary reading code. Quit when you get to a $00 end-of-glossary marker.

$2B2E–2B53 — MANAGE GLOSSARY STACK

If a match is found, increment the glossary stack pointer $2B92 by two and then set the glossary flag to read characters directly from the glossary rather than from the keyboard. If an attempt was made to nest the glossary entries more than eight deep, ring the ding dong and generate an error message. If the nesting depth is legal, save the present glossary pointer to the glossary stack as a lo-hi pair.

$2B54–2B7A — READ CHARACTER FROM GLOSSARY

Read the glossary stack pointer to find the current position in the glossary. Then increment the position pointer and force feed it into a command to get that character from the glossary file. Get and test that character. If a $00 (end of glossary) or an $8D (end of entry), then pop glossary stack. If a $DD or "]" fake carriage return, substitute a real $8D return. Exit with glossary character in accumulator.

$2B7B–2B91 — POP GLOSSARY STACK

Decrement the glossary stack pointer by two, and test the result. If you are >$00, this means you have nested glossary references, and are ready to return from callee to the caller. If <$00, you are done using the glossary and wish to return to normal text entry. To do this, cancel the glossary active flag and reset the nest pointer to $00.

$2B92 — GLOSSARY NEST POINTER

Points to an address pair in the glossary stack up to eight deep. Increments by twos each time an entry is nested; decrements by twos each time an entry is completely read.

$2B93–2BA2 — GLOSSARY NEST STACK

Holds up to eight pairs of pointers that show current "open" positions in the glossary.

$2BA3–2BCE — FILL GLOSSARY PROMPTER

Clear the screen. Set local pointer to start of glossary file at $1B00. Test for split screen and pick 8 or 16 lines for glossary display. Display the existing glossary selections. If not enough room, wait for user keypress. Repeat until all selections are shown. Then prompt for new entry and get user response.

$2BCF–2C12 — FILL GLOSSARY

Read user entry, aborting on $8D carriage return. If valid, replace the last $00 in the glossary with a $8D. Enter user string to glossary. Check remaining space in glossary, exiting with an error message if the glossary is full. Store a $00 at the end of the final glossary entry.

$2C13–2C24 — RETURN PROMPT

Displays screen message "Press return to exit."

$2C25–2C74 — AUXILIARY FUNCTION PROMPTER

Skip display if WPL is on. Turn off split screen and clear screen. Set local pointer pair $00–01 to point to the auxiliary functions menu. Put down the menu until the ending marker. Then put down the return exit prompt. Then display the selection prompt. Get user selection, force uppercase and range check A-K. Convert selection to a numeric, and double it to point to an address pair. Get the address pair wanted and put that pair into local pointer $80–81. Then do an indirect jump to the selected routine. Abort on any key except A-K.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$2C75–2CA3 — CONNECT KEYBOARD DIRECTLY TO PRINTER

Print [Q] exit prompt to screen. Find and set print destination to printer. Get one key and test for [Q] exit. If not, print character directly to printer. Also show on screen, flashing next cursed location. Continue until [Q] is pressed. On [Q], reconnect keyboard to word processing code and close any open files.

$2CA4–2CBE — QUIT APPLE WRITER

Put down quitting message and prompt. Get user key. If a "Y", destroy everything in sight and then some. Abort on any other key.

$2CBF–2D0E — PLOW THE SOUTH FORTY

A classic example of code with absolutely no redeeming social value. Fill the entire machine with zeros from $0300–FFFF except for the I/O space. Flip all soft switches to their normal positions. Then enter Applesoft without having any DOS available.

$2D0F–2D32 — CONVERT 1.1 FILE PROMPTER

Get user filename of Apple Writer 1.1 file. Splice together a DOS instruction of BLOAD TEXT.FILENAME, A$0800 and send this out, but hold off on the final $8D.

$2D33–2D61 — CONVERT 1.1. FILE SETUP

Try to move a copy of DOS to alternate memory, along with a DOS hook copy. Both of these attempts fail since alternate high memory is never accessed (see text). Send the spliced filename to alternate DOS, which BLOADS the 1.1 file into the usual LOFILE area. Note that this is a BLOADed binary file, rather than a standard DOS text file. Note also that both the DOS used and the results go into auxiliary RAM. At this point, the old 1.1 file is sitting in the right place, but still is coded wrong.

$2D62–2DB1 — CONVERT 1.1 CODE TO STANDARD ASCII CODE

Init the LOCURS pointer. Init HIFILE. Read the 1.1 file one character at a time, starting at $0801 and ending on a $60 marker. Recode each character to a standard ASCII text file character. Specifically:

$00 – 1F	→	$C0 – DF	capital letters
$20 – 7F	→	$A0 – FF	not used by 1.1
$80 – BF	→	$80 – BF	control characters
$C0 – DF	→	$E0 – FF	lowercase letters
$E0 – FF	→	$A0 – BF	numerals & punctuation

When finished, replace the end $60 with a $00. Update the screen and send cursor to beginning. Note that this conversion does not change imbedded commands. A separate WPL program called CONVERT is used for this.

$2DB2–2DE1 — SAVE GLOSSARY FILE

Reset append flag. Get filename, aborting on $8D carriage return. Set local pointer pair $80,81 to point to glossary file start at $1B00. Find glossary length and set LOCURS pointer to end of glossary. Save the old filename to the = buffer. Move present filename to DOS filename buffer. Close all open files. Open glossary save file. Save glossary. Close. Restore = filename. Update LOCURS.

$2DE3–2DF8 — FIND LENGTH OF GLOSSARY

Move up through the glossary at $1B00 one character at a time until the $00 end marker is found. Remember this address with a temporarily diverted LOCURS pointer.

$2DF9–2E29 — PRINT ONLY TO SCREEN DOS ACCESS

Clear the screen and reset the append flag. Get new filename and save old filename to = file. Move new filename to filename buffer. Open file. Read file to screen. Close file. Get old filename back from = file.

$2E2A–2E3B — LOAD GLOSSARY FILE

Get filename without slot and drive prompt. Abort on $8D carriage return. Close glossary nest. Trick DO routine into loading into the glossary at $1B00 rather than to the WPL program file. Load the file. Test for 2048 or fewer characters while moving file from DOS buffer to the glossary. Sound ding dong on overflow.

$2E3C–2E8D — GET FILENAME FROM USER

Update slot and drive prompt. Bypass screen prompts if WPL is active. Prompt screen for filename, then slot and drive if called for. Catalog disk on "?" filename, then try again. If an $8D carriage return, pop the stack twice to cancel both this sub and the calling sub. If a valid filename, force uppercase and save the filename to the local use buffer at $1A00.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$2E8E–2EA4 — **PRINT FILENAME TO DOS**

Read the filename from the local use buffer at $1A00. Force all uppercase. Output to COUT at $FDF0. Exit on carriage return, but do not output the carriage return.

$2EA5–2EC0 — **SAVE TAB FILE**

Get filename from user. Splice together filename from BSAVE TAB.USERNAME, A$1880, L$40 <cr>, and send command to DOS. Note that two passes through the reference file are used, keying on the $BF marker. Note also that the tab values are saved as a binary file and not as a text file.

$2EC1–2EDE — **LOAD TAB FILE**

Get filename from user. Splice together filename from BLOAD TAB.USERNAME, A$1880<cr>, and send command to DOS. Note that two passes through the reference file are used, keying on the $BF marker. Note also that the tab values are loaded from a binary file and not from a text file.

$2EDF–2EFA — **SAVE PRINT/PROGRAM FILE**

Get filename from user. Splice together filename from BSAVE PRT.USERNAME, A$18C0, L$140 <cr>, and send command to DOS. Note that two passes through the reference file are used, keying on the $BF marker. Note also that the print/program values are loaded from a binary file and not from a text file.

$2EFB–2F16 — **LOAD PRINT/PROGRAM FILE**

Get filename from user. Splice together filename from BLOAD PRT.USERNAME, $A18C0<cr>, and send command to DOS. Note that two passes are used through the reference file, keying on the $BF marker. Note also that the print/program values are loaded from a binary file and not from a text file.

$2F17–2F29 — **CREATE SYSTEM FILENAME**

Generates the print/program filename SYS and puts it in the local filename save. Then gets that print/program file. Used during init process to boot default PRT.SYS.

$2F2A–2F4E — **TAB PROMPTER**

Calculate present tab position. Clear and prompt screen bottom. Get user key. Force uppercase. If an S, set tabs. If a C, clear tabs. If a P, purge tabs. Abort on any other key. Update tab display.

$2F4F–2F6B — **CLEAR ONE TAB**

Scan through the tab file, looking for a position match on one of 32 tab pairs. If a tab has been set to this position, zero both parts of address pair.

$2F6C–2F76 — **PURGE ALL TABS**

Zero all 32 locations in the tab file.

$2F77–2FA9 — **SET ONE TAB**

Scan the tab file backward to see if this tab is already set. If the tab needs to be set find the lowest $00 pair in the tab file, and then set that pair to the present tab position. Should all 32 tab pairs already be in use, ring the ding dong.

$2FAA–2FE9 — **SEE IF TAB IS POSSIBLE**

Calculate the present tab position. Scan through the tab file, finding the next highest tab location if one exists. If no tab exists, update the screen and exit.

$2FEA–3019 — **TAB TEXT FILE**

Test solid-apple key for skip over. If no skip over, calculate how many spaces are needed between present position and tab position. Then enter those spaces to LOFILE. If skip over, calculate how many characters are to be skipped over, and move those characters from HIFILE to LOFILE, one at a time as needed.

$301A–3045 — **UPDATE TAB STATUS DISPLAY**

Clear each character in the tab status display by forcing it to normal, high ASCII. Go through the tab file and inverse all live tabs found, but ignore any tabs beyond 80 characters.

$3046–3095 — **TAB STATUS DISPLAY**

A file of 80 characters used for the tab display. Untabbed locations are normal text. Tabbed locations are inverse text. Note that this is a file that aliases badly if you try to disassemble it.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$3096–30AA — INIT PROCESSING FLAGS

Init LOFILE. Init HIFILE. Use normal data line display. Use normal wraparound. Turn data direction to ">". Turn off case changer. Turn off verbatim flag. Note that this code aliases easily. The correct disassembly should read:

```
$3096–  20 FF 26 JSR   $26FF
$3099–  20 25 27 JSR   $2725
$309C–  A9 80    LDA   #$80
```

Be sure to correct this detail before attempting to capture source code.

$30AB–3111 — MAIN PROCESSOR SERVICE ROUTINE

Init the stack pointer to $01FF. Init processing flags. Turn off busy prompt. Reset keyboard strobe. Turn display off. Update HICURS and LOCURS pointers. Write file to screen. Set data direction to ">". Unsplit screen. Verify DOS. If DOS 3.3e, do nothing. If ProDOS, do STARTUP. Turn display on. Test for print/program command and process if found. Turn off the flags that load only to screen, copy from memory, load from $A–$D flags. Use main DOS. If type-ahead buffer is empty and WPL is not active, update the screen. If WPL is active, use WPL file as $A–$D string source. If WPL is not active, get user input. If an escape, stop WPL. If no subroutine call also stop WPL.

$3112–3136 — PROCESS USER INPUT

Get user key from type-ahead buffer. Test for tutorial help needed, and provide if called for. If glossary is still active, continue getting characters from glossary till finished. Filter user response. Do user action. Repeat main service loop.

$3137–315F — FILTER KEYSTROKES

Test for a printing character. If real, put that character into LOFILE and update the screen. If a control character, process further.

$3160–31C2 — FILTER CONTROL COMMANDS

If an escape, toggle the data line display. Turn replace flag and mystery flag off. Set local pointer to point at [ ] prompt list. Scan the list for a bracketed letter. Quit if no match found. Shut off the case changer unless a case command. On a match, get the address pair from the function address list, put that pair into the local pointer and do a forced jump to the selected control routine.

$31C3–31F0 — SPLIT SCREEN PROMPTER

Clear and prompt bottom of the screen if WPL is not active. Get user response and force uppercase. If a Y, then split screen. If an N, then stop screen split. If a carriage return, then move live cursor to the other half of the split screen display. Ignore all other characters.

$31F1–31FB — CHANGE TO OTHER SPLIT SCREEN

If the screen is split, change the split screen flag to allow live entry on the other half of the display. Then update split-screen display values.

$31FC–320A — TURN SPLIT SCREEN ON

Activate split-screen flag. Init the split-screen pointer to LOCURS. Then update the split-screen display values.

$320B–3216 — TURN SPLIT SCREEN OFF

If WPL is not active, reset the split-screen flag by zeroing $F8. Then update cursors for full screen display.

$3217–3244 — UPDATE SPLIT-SCREEN VALUES

Update screen. Exit if not split screen. If split screen, calculate screen window. Save inactive split-screen pointer to address stash. Transfer LOCURS to active screen pointer.

$3245–325A — CALCULATE SCREEN WINDOW

If upper split screen, set window top at $00 and window bottom at $0C. If lower split screen, use $0C and $18. If full screen, use $00 and $18. Set cursor horizontal to extreme left. BASH the vertical address.

$325B–3265 — TOGGLE DATA LINE DISPLAY

Advance the $E5 data display flag to its next of three possible values, with $00 being no display, $80 being the usual MEM-LEN-POS display, and $C0 being the tab display.

$3266–326C — TOGGLE WRAPAROUND MODE

Change the $E1 wraparound flag either from or to its $00 broken-word or its $FF whole-word value.

$326D–3273 — TOGGLE CARRIAGE RETURN DISPLAY

Change the $D2 carriage return flag either from or to its $00 normal display or its $FF show returns as "]" display.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$3274–327A — **TOGGLE VERBATIM MODE**

Change the $D0 verbatim flag either from or to its $00 normal use of control characters or the $FF imbed control characters in text mode.

$327B–3280 — **PROMPT SCREEN BOTTOM AND GET RESPONSE**

Clear screen bottom, then prompt user. Get user response. Used when a filename is not involved.

$3281–32BF — **PROMPT SCREEN BOTTOM AND GET FILENAME**

Clear screen bottom, then prompt user for filename. Get filename. Abort on "?" or carriage return. If an "=" filename, append only slot and drive to old name, keying on $AC comma. If a new filename, transfer entire name to filename buffer at $0280. Find length of filename and save length to stash $E9. Clear remainder of filename buffer to all zeros.

$32C0–32D8 — **PROMPT SCREEN BOTTOM**

Clear bottom of screen. Print any user prompt found in the function list. Then print an ending colon. For instance, on a [S] command, the prompt "[S]ave:" is printed to screen.

$32D9–32EC — **CALCULATE SCREEN PROMPT POSITION**

Set horizontal position to extreme left. Change vertical position to line decimal 9 for a prompt on the upper split screen, or line 21 on the full screen or the bottom split screen. Then BASH this screen address.

$32ED–32FB — **CLEAR BOTTOM OF SCREEN**

Calculate screen prompt position. If WPL is not active, clear bottom of screen window. Advance vertical position for one blank line before prompt. Then BASH this screen address.

$32FC–3313 — **SAVE PROMPTER**

Set DOS clone flag to use auxiliary DOS in auxiliary memory. (This does not happen — see text.) Move old filename to "=" filename. Clear and prompt screen bottom and get a new filename. If a "?", then catalog the disk. When finished try again. If a carriage return, abort. If a legal filename, then fall through to the save filter.

$3314–3359 — **SAVE FILTER**

Set string source flag to use already entered filename. Turn off append flag. Read last character of filename for an $AB or "+" for append. If append, then set append flag $E2 and replace $AB with $00. Move LOCURS into the endpoint stash $98–99. Scan the filename for delimiters. If none are found, save entire text file. If delimiters are present, process them further.

$335A–3373 — **SAVE ENTIRE TEXT FILE**

Move cursor to end, putting everything into LOFILE. Set local pointer to LOFILE start at $0801. Send OPEN or APPEND command to DOS as needed. Save text file bytes to disk. Update screen. Reset string source flag to new entry by zeroing $AD.

$3374–33D5 — **INTERPRET SAVE DELIMITERS**

Set up special delimiters. Zero final delimiter. Set local pointer to LOFILE start. Begin moving characters from HIFILE to LOFILE, searching for the starting match string. Replace fake carriage returns with real ones in the search string and ignore wildcards. If no match is found, close open files, update screen, and replace "=" filename. If match is found, save the text file bytes to disk, using auxiliary DOS for auxiliary memory. Reset string source flag to new entry by zeroing $AD. Close any open files. Update screen.

$33D6–3455 — **SAVE BYTES TO DISK VIA RWTS**

Close open files. Clear bottom of screen. Open or append per state of $E2. Output filename, all capitals. Find the address of the DOS file manager parameter list and save as local pointer $00,01. Move DOS buffer addresses to file manager. Issue write and range-of-bytes commands to the file manager. Calculate the number of bytes to be written by subtracting local pointer from LOCURS. Feed this to the file manager. Feed the starting address to the file manager. Access DOS via RWTS. If no DOS errors, close open file. If an error, exit via error message routine.

$3456–3462 — **ACCESS DOS VIA RWTS**

Close open files. Test DOS flag. If $7F is a $00, then access main DOS in main memory. If $7F is a $FF, then try to clone a copy of DOS to auxiliary memory and access it. (The cloning fails — see text.)

$3463–3480 — **CLONE DOS TO AUX MEMORY (does NOT work!)**

Try to copy the image of DOS and the high monitor from $D000–FFFF in main memory into the same range in auxiliary memory. The attempt fails, since auxiliary high memory only switches on an alternate page zero switch. All attempts at accessing "main" or "auxiliary" DOS end up reaching the DOS in main high RAM. This module may be replaced with a single RTS.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$3481–3499 — PRINT FILENAME AS ALL CAPS

Read the active filename from the $0280 filename buffer, convert it to all capital letters, and send it to DOS. Include any ending trailers or carriage returns, but stop on a $00 end marker.

$349A–34BF — GET CHARACTER FOR MEMORY TO MEMORY LOAD

Increment $80 pointer. If $80 local pointer gets to LOCURS, change to HICURS and continue, scanning the entire file as needed. Read file value from either LOFILE or HIFILE. Force high ASCII. If at end of HIFILE, replace $FF marker with $00.

$34C0–34D0 — SET POINTERS FOR MEMORY TO MEMORY LOAD

Set $80 local pointer to LOFILE start at $0801. Set $82 local pointer to LOCURS.

$34D1–3506 — OPEN AND READ DOS FILE

If a memory-to-memory load, set pointers instead and exit. If a DOS load, close open files and clear bottom of screen. Alter the OPEN command so it does not create a nonexistent filename. Print the OPEN command to DOS, followed by the filename in all capitals, followed by a $8D carriage return. Unalter the OPEN command so it does create a nonexistent filename. Set up IOB. Read DOS file to buffer. Close file.

$3507–351F — SETUP SPECIAL DELIMITERS

If the usual "/" delimiter that does not allow fancy stuff, put a $01 into $EC, the "any length" stash, a $02 into $EB, the "fake carriage return" stash, and a $03 into $EA, the "wildcard" stash. If a special delimiter is in use, put the next higher ASCII character into $EC, the next one after that into $EB, and the one following into $EA. For instance, a "<" delimiter will have an "any length" symbol of "=", a "fake carriage return" symbol of ">", and a "wildcard" symbol of "?". Note that these four ASCII characters follow each other in sequential numeric order.

$3520–359F — CATALOG DISK

Replace "?" with space in filename. Change output hooks to display to the screen. Reconnect DOS hooks. Read keybuffer to see if a "#" command is present. If so, set local flag $80 to enter the catalog to the text file instead of showing it on the screen. Replace "#" with space if used. Unsplit and clear screen if not WPL. Put down prompt if catalog to text file. Save any slot and drive trailer to $1A00. Send all of a catalog command except the final carriage return to DOS, getting the command from the reference files and the slot and drive from $1A00. Test the catalog-to-text file flag. If to text file only, grab the screen hooks. If to screen, leave the hooks the way they were. Catalog the disk by sending out the final carriage return. Prompt the screen. Reconnect the hooks. Get user keypress. Update the screen.

$35A0–35E0 — LOAD PROMPTER

Set the load pointer to LOCURS. Set string flag to use old string. Save old filename to "=" file. Set normal case, load instead of append, and include delimiters. Get filename. Test for a "\" for screen-only load. If screen only, set $FD flag and erase the reverse slash to a $00. Test first letter in filename for a "?". Catalog disk and try again if catalog is needed. Abort on a carriage return. On a good filename, fall through to load interpreter.

$35E1–3663 — INTERPRET LOAD DELIMITERS

Zero the three delimiter stashes and test for the copy-from-memory "#", setting $D5 if present. Search through filename for any punctuation except for comma, period, or semicolon. If special delimiter punctuation is found, then setup special delimiters. Then test and set the all-occurrence $E2 flag and then test and set the omit-delimiters $0D flag. Open and read the DOS file, moving the first 256 characters into the DOS buffer at $0D00.

$3664–367A — SCAN DOS FILE FOR MATCH

Read one character at a time from the DOS buffer to the line buffer at $1600. Read the first character beyond the first load delimiter in the $0280 command file. If a wildcard, ignore and get the next DOS character. If a carriage return prompt, substitute for the carriage return. Compare the characters for a match. If a match, continue matching characters in the string against the DOS file until the second delimiter. If no match, keep searching.

$367B–36F1 — LOAD DELIMITED FILE PORTION

Test $0D to see whether delimiters are to be included. Begin transferring characters from the $1600 line buffer into the main text file, entering just above LOCURS. Include or exclude the delimiters per the $0D flag. Continue entering and transferring characters until a matching end string arrives. Cease entry either on a string match or the end of the DOS file. Note that additional DOS characters are read as needed to the $1D00 DOS buffer in blocks of 256.

$36F2–36FD — ALL OCCURRENCES LOAD PROCESSOR

If all occurrence flag is set, output a carriage return and continue the search and match process. If not set, fall through to next module.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$36FE–3714 — END OF LOAD CLEANUP

If load was to screen only, enter a press return prompt. If filename was not the "=" filename, get the old filename back from the $1A40 "=" file and put it in the active filename file at $0280. Then update the screen.

$3717–3751 — FIND PROMPTER

Clear the case flag $C4 and the all flag $E2. Set use old string flag $AD. Clear and prompt the screen bottom. Read first character in keybuffer, aborting on a carriage return. If an "=", get the old find string from the find save stash $1A80, and move it to the keybuffer. Move a copy of the keybuffer back into $1A80, saving what you now have as a future "=" string for future finds. Read the first delimiter and process special delimiters.

$3752–378E — INTERPRET FIND DELIMITERS

Zero the second and third delimiter stashes $A2 and $A3. Scan the find string in the keybuffer for delimiters. Put the position of the second delimiter into $A2. If there, put the position of the third delimiter into $A3. Read the character beyond the last delimiter. If an uppercase or lowercase "A" and if we are to replace, then set the all occurrence flag $E2.

$378F–3835 — SEARCH TEXT FOR FIND MATCH

Set the $98,99 search pointer pair to the present cursor position. Begin searching. Check keybuffer for an escape and quit if present. Zero the any length flag $C9. Read and hold the match character, setting the any length flag $C9 on an any length delimiter. If a wildcard, skip the match for this character. If a fake carriage return, replace with a real $8D. Compare the match character against the cursed character in the text file. If a match, then continue trying for a match on successive characters. If no match, continue the search, going in the direction set by the data direction stash $CF.

$3836–385D — FIND FOUND PROCESSOR

Get user instructions. If search only and a carriage return, continue the search. If any other character, abort. If search and replace as set by a nonzero $A3, then do the replacement on a "Y" or [Y] command, continue the search on a $8D carriage return, or abort on any other character.

$385E–3884 — PROMPT USER ON FIND

Update the screen. If not WPL, then clear screen bottom and put down the find prompt. If replace, then put down the "/ Y = Replace" trailer as well. Get user response and convert it to a low ASCII control character. Note that this gives both the user and WPL identical access to the find-and-replace code.

$3885–3886 — FIND REGISTER STASH

The Y register gets stashed to $3885, and the X register is held in $3886 during find matching.

$3887–38E1 — DO REPLACEMENTS

Move the characters to be replaced from HIFILE to the line justify buffer $1600. Test for a replace with nothing. If a replacement is needed, go through the replace string in the keybuffer and enter the needed characters to the text file. If a fake carriage return, substitute a real $8D. Update screen when replacement is complete.

$38E2–3902 — ADJUST REPLACE POINTER

If the find pointer is pointing to LOFILE, enter character to LOFILE. If the find pointer is pointing to HIFILE, then move a character from HIFILE to LOFILE. Used to allow replacement in either direction.

$3903–392B — NEW PROMPTER

Bypass screen prompting if WPL is active. Prompt screen and get user response. Force uppercase. If a "Y", re-init all word processing pointers, and reset HICURS and LOCURS. Zero the old filename. If not a "Y", abort.

$392C–3932 — FORCE UPPERCASE

Verify that the character is lowercase. Then subtract $20 to convert to uppercase ASCII. Abort on nonlowercase characters.

$3933–3941 — MEMORY FULL ERROR RECOVERY

If WPL was active, re-init the WPL flags. Set flag to reuse old string. Ring the ding dong. Delete last character.

$3942–3952 — CASE CHANGER

Set the case change flag $E8. Read the case flag $C4. If mixed case $00 then force uppercase $C0. If uppercase, then switch to lowercase $80. If lowercase, then switch to uppercase only.

$3953–3959 — DATA DIRECTION TOGGLE

Read the data direction flag $CF and change it, either to or from the $00 "<" or the $FF ">" state.

$395A–3960 — REPLACE MODE TOGGLE

Read the replace mode flag $F5 and change it, either to or from the $00 normal-entry or the $FF type-over mode.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$3961–398F — **CALCULATE CURRENT TAB POSITION**

Set the $80 local pointer to LOCURS and search backwards through the file for the next previous carriage return or else the start of the file. Count characters into $80,81 as you do so. Then subtract the last carriage return position from the LOCURS position, and save this to the tab pointer pair $96,97.

$3990–39F2 — **STATUS LINE SETUP**

Test status line flag $E5 to see which status line to use. If normal, verify 80-character line and put down fixed parts of "Mem-Len-Pos-Tab" header all on top line in inverse. Home the cursor to top of screen for normal and upper split displays, to 12 lines down for the lower split display option. Check the data direction in $CF and the status of case flag $C4. Put a "<" or a ">" in the first display slot if the case flag is not active, a "U" or an "L" if it is. Note that this first display location is the same on both normal and tab status lines.

$39F3–3A05 — **DISPLAY TAB STATUS LINE**

Test the status line flag $E5. If a tab display is needed, then move the tab status display to the correct line, knock a line off the available display space, and BASH the address. Skip for normal status line display.

$3A06–3A45 — **DISPLAY NORMAL STATUS LINE**

Put a space after the data direction slot. Test the verbatim flag $D0 and the replace flag $F5 and put an inverse space for nothing, an inverse "V" for verbatim, and an inverse"R" for replace. Note that you cannot get into both the V and R modes at the same time. Test the busy stash at $F4, and enter an inverse space if not busy and an inverse "*" if busy. Test the wraparound flag and enter an inverse space if words are broken or a "Z" if they are not. Calculate the memory remaining by subtracting LOCURS from HICURS, and save to the $9A,9B memory left stash. Tab over the "MEM:" display. Convert the memory remaining from hex to decimal, and display it.

$3A46–3ABA — **CONTINUE DISPLAY OF NORMAL STATUS LINE**

Find total available text file memory size and subtract two, allowing for the $FF starting and ending markers. Subtract remaining memory, convert to decimal, and display as length of current text file. Subtract LOMEM from LOCURS and knock two off for start and end markers. Convert this to decimal and display as position. Get the tab value, convert to decimal, and display as tab. Note that the Y register sets the horizontal position and that each prompt is skipped over so that each numeric fits where it belongs. Skip over the file prompt. Get the filename, force uppercase, and display the first 22 filename characters in inverse. Pad inverse spaces to the end of the screen. Decrement the available number of display lines by one and move down one line so that the screen update does not overwrite the status line. BASH the address.

$3ABB–3ACB — **DISPLAY DECIMAL VALUE**

For up to five digits, read the already converted decimal value in the $16F0 file. Replace any leading zeros with inverse spaces, thus right justifying.

$3ACC–3AD5 — **STORE TO SCREEN IN INVERSE**

Force the character in the accumulator to inverse by ANDing to low ASCII. Put on screen.

$3AD6–3B41 — **NO COMMENT**

Use and purpose of this code block is left as an exercise for the dedicated student. Many volumes could be written on this.

$3B42–3B66 — **FORCE FILE TO HIGH ASCII**

Scan LOFILE or a portion of LOFILE. Force all characters to high ASCII. Clears any end-of-screen line markers that may be present.

$3B67–3B76 — **FIND START OF SCREEN LINE**

Scan backward through the text file until the first low ASCII character, designating the end of the previous screen line, is found. Abort on $FF start of file.

$3B77–3B7D — **FORMAT SCREEN LINES LINK**

Format screen lines starting with load pointer instead of LOCURS. Fall through to next module.

$3B7E–3B99 — **FORMAT SCREEN LINES SETUP**

The end of each screen line to be displayed is marked in the text file with a low ASCII character, providing whole word breaks and giving the screen update code a way of knowing how far back in the text file to go to begin display, such that the cursor usually stays on the center screen line. Starting with the LOCURS pointer, back up two existing screen lines. Then reformat all lines to the end of the file, removing any previous low ASCII characters and putting new ones only at the end of each line to be displayed.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$3B9A–3C3C — **FORMAT SCREEN LINES**

Save present start of screen line to $BA,BB local pointer. Allow 79 characters per line, saving room for a possible "]" carriage return display. Scan the text file for characters. If you run out of LOFILE, then switch to HIFILE and keep scanning. Find out how many whole words will fit on a line, including multiple spaces and stopping on 79 characters, a carriage return, or an $FF end of HIFILE. Save the actual line length to stash $C5. Using this value, scan the line in the text file once again, and replace all low ASCII characters that may be left over from a previous formatting into high ASCII. When you get to the end of the line, force the last character to low ASCII, marking the end of the line. Repeat for as many lines as are left in the text file.

$3C3D–3C5E — **CASE CHANGER**

Force low ASCII, saving MSB. Ignore if a numeric, a punctuation mark, or a control command. If a lowercase letter from a–z and if in "U" mode, subtract $20 forcing uppercase. If an uppercase letter from A–Z and if in "L" mode, add $20 forcing lowercase. Restore MSB.

$3C5F–3CFE — **UPDATE SCREEN**

Abort if no screen display flag $F7 is set. Set the screen start pointer to a point in LOFILE some 12 lines previous for full screen and 6 lines previous for split screen. Set bottom of screen limit to $C8. Home the cursor and test data line flag $E5. Put down normal or tab data line as needed. Begin getting file values from LOFILE and putting them on the screen. If the wraparound toggle is set, break on whole words. Note that the final character in each screen line is a low ASCII value in the text file. Use this coding to force whole word breaks, or ignore it for continuous display. If the carriage return display flag is set, show each carriage return as a "]". Continue through LOCURS until its end. Then switch to HICURS and keep entering characters until the screen limit is reached.

$3CFF–3D13 — **SWITCH SCREEN POINTER TO HIFILE**

Save the horizontal and vertical screen changeover positions to $8A and $8B. Then update the screen pointer to the start of HIFILE. Note that LOFILE holds everything up to and including the cursor, while HIFILE holds everything beyond the cursor to the end of the file.

$3D14–3D44 — **STORE ONE CHARACTER TO 80-COLUMN SCREEN**

Force low ASCII if a control command. If a key is pressed and if WPL is not active, get the key. Verify 80 columns. Divide horizontal position by two, picking an even or odd location. If even, store character to even screen in main memory. If odd, store character to odd screen in auxiliary memory.

$3D45–3DA8 — **PRINT WPL ERROR MESSAGE TO SCREEN**

Save error number now in accumulator. Clear bottom of screen. Print error prompt to screen. Get error number back. Scan through the error message file, counting carriage returns. Stop when you get to the correct message. Print the error found. If error zero, LABEL NOT FOUND, go back to the WPL line being parsed in the keybuffer and print the label, stopping on the first space or control character. Print return prompt. Wait for user response.

$3DA9–3DC8 — **GET PRINT/PROGRAM VALUE FROM USER**

Show the print/program display values. Clear and prompt screen bottom. Get user string. If a $8D carriage return, abort. Save prompt pointer. Try to enter new print/program value. Ignore invalid user responses. Restore prompt pointer. Repeat until user enters a carriage return instead of a new print/program value. Note that this code is reentrant, calling its own callee. This allows prompting only when requested.

$3DC9–3DF4 — **PRINT/PROGRAM PROMPTER**

Clear and prompt screen bottom. Get user response. If "?", display current values and get new values from user. Save first two characters of user response and force both to uppercase. Usually, these will match a print/program value command. Save these characters to stashes $A2 and $A3. If the first keybuffer value is either a carriage return (user entry stop) or a $00 (comment line in WPL) then pop stack twice to undo re-entrant call to itself. Otherwise, back keybuffer up two characters to leave only the value or string following the print/program command. Then interpret and update the print/program command.

$3DF5–3E02 — **REMOVE PREFIX FROM KEYBUFFER**

Back each character in the keybuffer up by two, stopping on a carriage return. Used to eliminate a print/program prefix, leaving only the string or value to be entered.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$3E03–3E64 — **SUBSTITUTE (X), (Y), OR (Z) VALUES**

Move the keybuffer to the work buffer at $1A00, stopping on a carriage return. Scan the workbuffer for an "(X)", a "(Y)", or a "(Z)", carefully checking for leading and trailing parentheses as well as for range. Do this while moving characters back to the keybuffer so that, when substitution is needed, everything up to the (X) is back in the keybuffer. Take the letter found, subtract and then double so that X = $00, Y = $02, or Z = $04. Get the hexadecimal X, Y, or Z value from the print/program file and convert to decimal. Move up to five decimal digits to the keybuffer, ignoring leading zeros and left justifying. Continuing scanning the $1A00 work buffer, beginning one past the previous ")". Abort on carriage return.

$3E65–3E85 — **TEST FOR LEGAL WPL COMMAND**

If WPL is active, substitute for any (X), (Y), or (Z) values in the keybuffer. If the keybuffer holds a decimal value, convert it to hex. Scan the print constants match file and the WPL constants match file for a match against the two command letters in $A2 and $A3. If a match is found, go enter print/program value. Abort on no match.

$3E86–3EF1 — **ENTER PRINT/PROGRAM VALUES TO FILE**

Save the match command. Test command. If UT, a justify, or a WPL command, use next modules. Test the arithmetic mode on all commands except PM, which is always relative. If absolute, zero the old print/program value. Either way, add the old value to the new value and save. On absolute, the new value gets added to zero. On relative positive, the new value gets added to the old. On relative negative, the previously twos-complemented new value is added to the old value. If an SX, SY, or SZ relative command, set string source flag $AD. If a PN, move a copy of the PN value to the running page counter.

$3ED0–3EE4 — **CHANGE UNDERLINE TOKEN**

If the underline token is a space or a carriage return, zero the token. If any other character, save it to $19DE as the current token.

$3EE5–3EF1 — **PROCESS JUSTIFY FLAG**

Convert the match counter value such that FJ = $00; LJ = $01; RJ = $02; and CJ = $03. Save the justify mode to $19E0.

$3EF2–3F05 — **EXECUTE A WPL COMMAND**

Change the match command so that it points to a WPL command address by subtracting $28. Get the WPL command address pair and stash it in local pointer $82,83. Then jump indirectly to the command and execute it.

$3F06–3F32 — **WPL COMPARE STRING**

Remove any leading spaces from the keybuffer. Save the first delimiter to local stash $00. Find the second delimiter. Compare the characters one at a time until the third delimiter. If a perfect match, or if no string at all, set flag $AD to $FF.

$3F33–3F50 — **WPL SUBROUTINE CALL**

Test for a nesting of less than 32 subs, exiting via an error message if exceeded. Save the WPL program counter to the WPL stack and increment the WPL stack pointer by two. Then GO and execute at the point in the WPL program where you have a label match.

$3F51–3F68 — **WPL SUBROUTINE RETURN**

Test the WPL stack pointer to make sure a return address exists. If not, exit with error message "RT WITHOUT SR". Knock one count off the WPL stack pointer and then put the return address pair from the WPL stack into the WPL program counter.

$3F69–3F6C — **TURN VIDEO DISPLAY ON OR OFF**

On a YD command, store a $00 to flag $F7 turning the video display on. On an ND command, store a $FF to flag $F7, turning the video display off. WPL programs run much faster with the display off.

$3F6D–3FA2 — **WPL LOAD STRING**

Identify WPL string $A–$D to be used. Using the [L] command, move a copy of what is to be loaded above LOCURS in LOFILE. Then move the copy to the keybuffer, forcing high ASCII and providing an ending carriage return. Erase the copy above LOCURS to all zeros, and restore LOCURS. One more time: The "empty area" between LOFILE and HIFILE is temporarily borrowed to ease memory management on a WPL string load, letting you load from disk or memory.

$3FA3–3FD1 — **WPL STRING IDENTIFIER**

Scan the keybuffer for an "=$A" through "=$D" command, verifying both prefix and range. Convert the A–D ASCII character to a 0–3 by subtraction, and then multiply by $40. Note that it is faster to right shift on a six-bit multiply. Save the result to local stash $02 such that string $A = $00, $B = $40, $C = $80, and $D = $C0. Erase the command by replacing the equals with a carriage return.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$3FD2–4016 — **SET PRINT DESTINATION**

If PD = $08, then print to disk by getting the filename and storing $FDED to the print destination pointer $9E,9F. If PD = $00 then print to screen by setting the print destination pointer to $260B. If PD = $01 through $07, then send a NUL command to activate the printer card, and save the destination to $9E,9F. With a normal printer in slot one, the destination is $C100.

$4017–409A — **DISPLAY PRINT/PROGRAM VALUES TO SCREEN**

Unsplit and clear the screen. Set local pointer pair $80,81 to the print/program display image. Note that each prompt ends with a $BD marker. Print the first 11 print/program values using the subroutine below. Next, get the needed print/program value from the $19C0 file, changing first to decimal and then printing the decimal value to the screen. After doing 11 values, advance the pointer to the $19C0 print/program value file by 6 to skip over the WPL numeric values. Note that adding five with a set carry adds six. Then print the UT value. Then use the justify value of $00–03 to pick a letter pair of FJ through CJ, and print that letter pair. Print the top line TL from its $18C0 file, followed by a carriage return. Do the same for the BL file at $1940. Put down a return-exit prompt.

$409B–409F — **PRINT A CARRIAGE RETURN TO THE SCREEN**

Get an $8D carriage return and send to screen, moving you to the start of the next display line.

$40A0–40B4 — **PRINT ONE PRINT/PROGRAM PROMPT TO SCREEN**

Print the next portion of the print/program display image to the screen, stopping on a $BD marker. This puts down the fixed part of each message. Add a space to the end of each message.

$40B5–40C1 — **SAVE OLD FILENAME TO = BUFFER**

Get the old filename from the active filename buffer at $0280 and save it to the $1A40 "=" filename buffer. Note that this is always a temporary and local save.

$40C2–40CE — **RESTORE OLD FILENAME FROM = BUFFER**

Get old filename from the "=" buffer at $1A40 and put it back into the active filename buffer at $0280.

$40CF–40D5 — **LOAD GLOSSARY FILE**

Clear the glossary nest. Then trick the WPL file loader below into loading to the glossary at $1B00 instead. Bypass any WPL setup as you do this.

$40D6–412D — **ACTIVATE WPL AND LOAD DO FILE**

Set the WPL flag $DF to activate WPL. Set the WPL program counter to $0E00. If really WPL and not a glossary load, zero the WPL continue flag $E7, the string $A–$D activity flag $F6, the WPL subroutine stack pointer $92, and the string source flag $AD. Save old filename. Get DO filename. Open DO file. Verify DOS 3.3e. Get old filename back. Set destination address of file to $0E00 for WPL or to $1B00 for glossary file. Read the DOS buffer and load a maximum of 2048 characters into the selected file. Exit with an error message if too many characters are read. Put a carriage return both at the end of the file and into the WPL current character stash $8C. Add a zero ending marker to the selected file.

$412E–4143 — **MOVE FILENAME TO ACTIVE FILENAME BUFFER**

Move the filename in the $0200 keybuffer into the active filename buffer at $0280, forcing uppercase and exiting on a carriage return. Replace the carriage return with a $00 marker at the end of the filename.

$4144–4172 — **INTERPRET WPL COMMAND**

Check keybuffer for ESC key. If present, quit WPL. If string flag is active, get string character from $A–$D string file. Clear apple key stash. Test for end of WPL command carriage return. If not, read next WPL character. If it was the end of a command, skip over any possible label on the next WPL line. Remove all padding spaces as well, aborting on an $FF ending. This leaves the WPL program counter pointing to the first command character in the next WPL line.

$4173–4195 — **READ ONE WPL CHARACTER**

Read the next WPL character. If a $00, then quit WPL. Increment the WPL program counter and save the read character to $8C. If the previous character was an "=" and this character is a "$" and if the next character is a $A–$D, then initialize a string read.

$4196–41A3 — **INITIALIZE STRING $A–$D READ**

Convert the $A–$D string character into a pointer by subtracting $C1 and multiplying by $40. Note that a six-bit multiply is faster when right rotating. Save the result to string pointer $8E so that it points to the start of the correct string, with the $A offset of $00, $B offset of $40, $C offset of $80, and $D offset of $C0. Increment the WPL program counter and activate the $A–$D string activity flag $F6. Fall through and read the first string character.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$41A4–41B2	— **READ CHARACTER FROM $A–$D STRING** Read the current character in the open $A–$D string, and hold in accumulator. If a $00, turn off the string activity flag $F6. Advance the $8E string pointer.
$41B3–41CB	— **QUIT WPL** Put a $00 in the first WPL program slot to kill old program. Turn WPL off by clearing MSB of $DF. Reset the key strobe. Turn the display on. Empty the type-ahead buffer by forcing the $F3 emptier to match the $F2 filler.
$41CC–41CF	— **ENTER BOTTOM LINE** Trick the top-line entry code to use the bottom line by adding $80 to the pointer.
$41D0–41EF	— **ENTER TOP LINE** Move the keybuffer to either the top-line buffer beginning at $18C0 or to the bottom-line buffer starting at $1940. Zero the end of the string. Count characters as you go along, sounding ding dong, and outputting an error message if >128 characters. Note that these TL and BL buffers hold their line in "compact" form with delimiters instead of spaces and with a "#" instead of the page number.
$41F0–41FD	— **PAD BOTTOM SPACES** Compare the printed line counter against the page interval PI. Output a space and a carriage return as often as needed to reach the bottom of the page.
$41FE–420B	— **TEST CONDITIONAL FORM FEED** If a zero value in the hexadecimal stash, do unconditional form feed. If a nonzero value, find out how many lines are left on the page by subtracting the line counter from the last line stash. If a form feed is needed, fall through and do it.
$420C–421A	— **DO UNCONDITIONAL FORM FEED** Print repeated spaces and carriage returns until the line counter matches the last line stash. Set form feed flag $FF.
$421B–4224	— **PRINT SPACE AND CARRIAGE RETURN** Print one space followed by one carriage return, first using the print subroutine as a sub for the space and then jumping to it for the carriage return. NOTE: Introduces a possible "page creep" bug on top and bottom headers if both Apple Writer IIe and the printer card are set to 80 columns.
$4225–4273	— **WPL JUMP OR GO** Abort if WPL is not active. Find actual start of label in keybuffer, ignoring leading spaces. Save start pointer to $05. Start at the beginning of the WPL program file, and look just beyond each carriage return for a match to the first label character. On a first character match, continue matching until a space or else a control character marking the label end. If no complete match, scan WPL program file some more. If no match at all, exit with LABEL NOT FOUND error message. If match found, exit with WPL program counter pointing to labeled line and with an $8D in the WPL current character save.
$4274–427B	— **WPL PRINT TO SCREEN** Print keybuffer to screen until either a carriage return or a "=$". Then print an ending carriage return to the screen.
$427C–4299	— **PRINT KEYBUFFER TO SCREEN** Scan the keybuffer, printing one character at a time to the screen. Abort on either a carriage return or on the "=$" that is needed by the IN command.
$429A–42DA	— **WPL PROMPT AND GET STRING** Clear bottom of screen. Put down WPL prompting message in keybuffer, up to the "=$". Verify the "=", then run an $A–D range check. If a valid command, decrement the string assignment flag $03 and convert the A–D into an offset of A = $00, B = $40, C = $80, D = $C0. Save offset in local stash $02. Then get the user response, carefully isolating that response from WPL and glossary processing by temporarily disabling the $DF flag. At this point, the user response will be in the keybuffer, but the assign string flag will only be activated on a legal string assignment. Test the assign string flag. If set, fall through to string assignment. If cleared, ignore.
$42DB–4304	— **WPL ASSIGN STRING** If a new string assignment, use the WPL string identifier subroutine to calculate offset and verify legality. Calculate the address of the 64-character-length limit for the string in use, saving to local counter $00. Read the string in the keybuffer and transfer to the correct string file with $A starting at $1780, $B at $17C0, $C starting at $1800, and $D starting at $1840. Do this until a carriage return. Replace the end of string carriage return with a $00 marker. If string is too long, exit via a STRING OVERFLOW error message.
$4305–4313	— **PROMPT FOR SINGLE PAGE** Turn off the string assigner flag $03. Put the "INSERT SHEET, PRESS RETURN" prompt onto the screen.

Table 12-7—cont. Detailed Script of Apple Writer IIe Main Program

$4314–433F	**— SETUP TL OR BL PRINTING** For BL, use an offset of $80 and the $1940 file. For TL, use an offset of $00 and the $18C0 file. Calculate the line length RM - LM and save to $D3. Save the old underline status. Force no underline. Fill the formatting buffer at $1A00 with spaces up to the required line length. Mark end of line with a $00 marker. Read first character of TL or BL file. If a $00, no TL or BL is in use, so restore underline mode and exit. If any other character, save as delimiter to $E6 and then fall through to the formatter.
$4340–438F	**— FORMAT TOP OR BOTTOM LINE** Note that the TL or BL is read from its compact form (delimiters and "#") buffer at $18C0 or $1940 and is then expanded (spaces and full page number) into the formatting buffer at $1A00–1AFF. The $0200 keybuffer is used as a temporary work area, to simplify entry of page numbers and the center and right-justification process. The keybuffer works in a "batch" mode where it first does the left justify text and moves it to the formatting buffer, followed by the center, and finally the right justified text. Should a "#" crop up at any time, it is replaced with the decimal page number. Switching from left to center to right takes place when the corresponding delimiter crops up. Pointer $00 keeps track of repeat trips through the same code, with $00 = left, $01 = centered, and $02 = right. Entry of the left string starts at the left margin. The center string starts half the number of center characters short of center. The right string starts the number of right characters short of extreme right.
$4390–43A6	**— PRINT FORMATTED TOP OR BOTTOM LINE** Pad the left margin with the needed number of spaces. Then print the top or bottom line as previously formatted in the $1A00 buffer. Continue until a $00 marker and then print a space and a carriage return. Restore the underline mode.
$43A7–43C0	**— GET DECIMAL PAGE COUNT** Read the hex value of the running page counter $BE,BF and convert it to decimal. Put the decimal result in the current position in the keybuffer, left justifying and dropping any leading zeros.
$43C1–43CA	**— WPL PRINTER ENABLE** Decrement the $B8 printer flag if not in the print-to-screen mode. Apparently undocumented and not used in this version.
$43CB–43D5	**— INIT PAGE NUMBER** Move a copy of the starting page number in $19CA and $19CB into the running page number counter $BE,BF.
$43D6–43E4	**— PRINT TOP MARGIN** Get top margin value from $19CA. Abort if zero. Print space and carriage return for each top margin line needed.
$43E5–43F6	**— PRINT BOTTOM MARGIN** Compare line counter against last line stash. Print space and carriage return for each line until they match. Then get top margin value from $19C8, and print a space and carriage return for each bottom margin line needed. Do this by moving "up" to the previous module.
$43F7–4413	**— SETUP NEW PRINTING** Init the running page number counter from the page number stash. Init page printing. Save LM and RM to temporary stash at $5212 and $5213. Turn off the underline mode. Then enable the printer. Fall through to page printer, but do not increment the page number.
$4414–4419	**— SETUP CONTINUE PRINTING** Init page printing. Then jump to the page printer, entering where the printing can continue on the middle of a page.
$441A–4435	**— INIT PAGE PRINTING** Set the first line in paragraph flag $DD. Reset the keystrobe. Update the print destination. Move cursor to end, thus putting entire text into LOFILE. Set printer pointer $90,91 to LOFILE start at $0801. Clear the cease-printing flag $D4.
$4436–448A	**— PRINT ONE PAGE** Add one to the running page number. If single page, prompt user and await reply. Zero the line counter $DA, the footnote buffer flag $FE, and the form feed flag $FF. Save the left margin to the left margin padding stash $B7. Get the number of printed lines and subtract the bottom margin. Knock another count off if there is an active bottom line. Save to the last line stash $DB. Format and print the top line. VTAB the top margin by printing spaces and carriage returns. Print the body of the page. If not to end of file and if not a conditional form feed, print the footnotes, the bottom margin, the bottom line entry, and the final bottom padding. If not to end of file, repeat for another page. If at end of file, fall through to cleanup.
$448B–44B0	**— END OF PRINTING CLEANUP** If WPL is not active and print is to the screen only, then put down return prompt. Send null $00 to printer. Display full screen. Move cursor to beginning. Connect screen as print destination. Close open files.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$44B1–44C1	—	**PRINT BODY OF PAGE** Test to see if the bottom of the page or the end of the file has been reached. If not, print another justified line. Repeat till page is full or file is empty.
$44C2–44CF	—	**FIX PARAGRAPH START** If this is the first line of a paragraph, add the paragraph margin to the left margin padding stash $B7. Note that this is always a relative addition, with negative page margins having been previously twos-complemented.
$44D0–44EC	—	**PRINT ONE JUSTIFIED LINE** Read the keyboard. If an escape, decrement the cease printing flag and clear WPL. If not, fix paragraph start. Test line for footnotes, underline, or imbedded printing commands. Abort if form feed flag is set. Format the line to be printed. Print that line.
$44ED–44F6	—	**GET ONE CHARACTER FROM LOFILE** Get one character from LOFILE and convert it to high ASCII. If not zero, save the character to the accumulator and exit. If $00, then set the cease printing flag $D4 and exit with an $8D carriage return in the accumulator.
$44F7–452B	—	**UNDERLINE DETECTOR** Calculate remaining line length RM - HPOS, saving to $D3. Get next character from file. Test for the underline token. If UT, then toggle underline mode. If end of line, then exit to footnote capturer. If a space, repeat until the first nonspace character.
$452C–4550	—	**FOOTNOTE DETECTOR** Save present line horizontal position. Get next character, exiting and setting the cease printing flag if end of file. Test for the "(< " start of footnote prefix. If found, save to keybuffer. If absent, fall through to imbedded command detector.
$4551–456A	—	**IMBEDDED COMMAND DETECTOR** If a "." and if the first character on the line, go process imbedded command. Reset the left margin. If form feed flag inactive, get another character via the underline detector.
$456B–4594	—	**SAVE FILTERED CHARACTER TO LINE BUFFER** At this point, any remaining characters are real and not part of a footnote or an imbedded printing command. Save the active character to the line buffer at $1600. If a carriage return, save the line length and set the first line in paragraph flag. Then go and justify line. If an underline token, adjust line length. If a space and the first character on the line, swallow the space. Advance the printer pointer $90,91.
$4595–45C8	—	**PRINT FOOTNOTE** Save underline status. Begin footnote with no underline. Test the footnote flag and abort if none in use. Set local pointer $00,01 to start of footnote buffer at $1200. Print space and carriage return. Pad left margin with spaces. Get the current footnote character. Exit on a $00 end marker. If not end, then print the character. If a carriage return, then pad left margin. Continue for all footnote characters. Restore underline status on exit.
$45C9–4605	—	**SAVE SINGLE FOOTNOTE TO KEYBUFFER** If the first footnote for this page, set footnote buffer flag $FE and knock two counts off the number of printed lines. If a second or higher footnote for this page, decrement the footnote buffer flag and knock one count off the number of printed lines. Save the footnote to the keybuffer, sounding the FOOTNOTE OVERFLOW error if >128 characters in the footnote. Continue entry of the footnote to the keybuffer until the ">)" end marker. End the footnote with a $00 marker in the keybuffer. Advance the print pointer by three to bypass footnote.
$4606–461F	—	**FIND END OF FOOTNOTE BUFFER** Set local-use pointer pair $00,01 to start of footnote buffer at $1200. Scan buffer until the $00 end marker, which is where the current footnote is to begin. If no $00 end marker within 2048 characters, then exit with FOOTNOTE OVERFLOW error message.
$4620–4632	—	**FOOTNOTE ERROR PROCESSOR** Clear WPL. Move cursor to beginning of full screen. Set print destination to screen. Close open files. Exit to WPL error message printer.
$4633–4635	—	**WARM RESTART LINK** Apparently unreferenced and unused in this version.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$4636–464C	— **MOVE SINGLE FOOTNOTE TO FOOTNOTE BUFFER** Begin moving characters from the keybuffer to the open end of the footnote buffer. If an $8D carriage return, then make room for the extra line by decrementing the last line stash $DB. Exit on a $00 end marker. If >128 characters with no end marker, then exit via a FOOTNOTE OVERFLOW error.
$464D–4678	— **PROCESS IMBEDDED PRINTING COMMAND** Get the first character of the printing command. If a $00 end of file, set the cease-printing flag. Save the character to the keybuffer. Repeat until a carriage return. This moves all of the imbedded command, except the leading period, into the keybuffer. If not at end of file, advance the printer pointer by one. Then do the imbedded print/program command.
$4679–4681	— **CALCULATE ROOM LEFT ON LINE** Subtract the current line position $B7 from the right margin stash $19C4 and save as room remaining stash $D3.
$4682–468B	— **CALCULATE LINE LENGTH** Subtract left-margin stash $5213 from the right-margin stash $5214, and save as room remaining stash $D3. Note that these saved values cause all top and bottom lines to have constant widths, regardless of margin changes imbedded in the text.
$468C–46BC	— **PRINT FORMATTED LINE** Dump the formatted line in $1600 to the printer, quitting when you get to a $00 marker, a carriage return, or if the line length stash $DC is exceeded. Quit if the line interval is zero. If spaces are to be added between lines, add them, but not beyond the bottom of the page.
$46BD–46C2	— **PAD LEFT MARGIN TO DEFAULT VALUE** Print as many spaces as are needed to create a left margin using the default left-margin position set up at start of printing.
$46C3–46D0	— **PAD LEFT MARGIN TO CURRENT VALUE** Print as many spaces as are needed to create a left margin using the current left-margin value as modified by PM or imbedded LM commands.
$46D1–46DA	— **PICK JUSTIFY MODE** If left justify, do nothing. If full justify, go and do full justify module. If RJ or CJ, fall through to the next module.
$46DB–4710	— **CENTER OR RIGHT JUSTIFY** Save justify mode to the Y register with $02 for right and $03 for center. Subtract the number of characters in the line from the line width. If center justify, divide by two to split the difference. Move the characters in the line buffer $1600 either to the extreme right or to the center, starting at the highest character and working backward. Then add spaces to every location before the formatted string. Note that spaces are not needed beyond the string, since a carriage return stops printing.
$4711–4774	— **FILL JUSTIFY** The fill justify mode works by adding extra spaces as needed to each existing space, always starting at the left, thus "expanding" the line to fit the available space. Reset the padding counter $C7. Skip adjusting first spaces if this is the first line in the paragraph. Count the number of nonspace characters in the line. Abort if no padding is needed. Add one space per space starting at the right by moving the text beyond the space one slot to the right in the $1600 line buffer. If an underline, then adjust for underline mode. Repeat padding until line is fully justified. Quit when padded line length equals available line length.
$4775–4782	— **ADVANCE PRINTER POINTER TO NEXT LINE** Add the length of the characters actually used during the last printed line to the printer pointer $90,91. This sets the pointer to the start of the next line, bypassing everything printed or used as footnotes or imbedded commands so far.
$4783–47E0	— **PREPARE ONE CHARACTER FOR PRINTING** Save registers. If a space, and in underline mode, save "__" underline symbol to $4802; otherwise save character to $4802 stash. Test for underline token UT. If present, toggle the underline flag and replace with a space. If a carriage return, increment the line counter $DA and output the carriage return. Also test CR stash and issue as many line feeds as needed for correct vertical spacing. Also restore registers and exit. If not a carriage return, check the underline flag. If underline is needed and if not a space, print "__" symbol, followed by a backspace, followed by the next character. Restore registers and exit.
$47E1–47E8	— **SEND CHARACTER TO PRINTER** If the continue printing flat $B8 is enabled, send the character out through the printer pointer $9E,9F. This pointer may point to a printer, DOS, or the screen as previously set up.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$47E9–47F2	— **ISSUE DOS PREFIX** Send an $8D carriage return followed by an $84 DOS attention "ctrl-D" to the $FDED output character hooks for DOS 3.3e access.
$47F3–4801	— **CLOSE OPEN FILES** Print the DOS prefix, followed by CLOSE to the $FDED character hooks for DOS 3.3e access.
$4802–4805	— **ALTER DOS VALID OPEN KEYWORD** Store the accumulator to the OPEN keyword for DOS 3.3e. This will activate the creation of a new file during the OPEN command if a $23 and will stop creation of a new file if a $22. Note that the real DOS address has been modified to $E123 since the DOS used is $3800 locations higher than stock. More details in Beneath Apple DOS.
$4806–480A	— **DOS I/O ERROR HANDLER** Force I/O error #8 and fall through to the error handler.
$480B–4833	— **DOS ERROR HANDLER** Reset OPEN keyword to allow creation of a new file. Find the DOS error number from DOS itself. Note the correct address is $E25C since DOS is installed $3800 higher in memory than stock. Clear WPL. Close glossary nest. Grab I/O hooks and verify 80 columns. Close open files. Clear bottom of screen. Cancel old file name. Verify DOS 3.3e. Clear to EOL. Fall through to error message printer.
$4834–4863	— **PRINT DOS ERROR MESSAGE TO SCREEN** Clear screen to EOL. Put down the "DOS:" prompt. Go into DOS 3.3e with the error number and get the error pointer. Note the correct force-fed addresses of $E23F and $E171, since DOS is $3800 higher than usual. Print the error message to the screen, followed by a space. Ring the ding dong. Get user response. Re-enter to the word processing main service loop.
$4864–488D	— **SETUP DOS IOB** Clone a copy of the existing DOS input-output block, or IOB, and move to $02E0. Zero the volume slot to accept any volume. Also zero the pointer to the DOS read buffer and set the pointer to the DOS T/S buffer to $FE. Move the current track and sector to the T/S buffer.
$488E–48A7	— **READ CHARACTER FROM FILE** Save registers. Test the $D5 memory source flag. If character is to be read from memory, do it. If character is to be read from DOS, do so. Set the carry flag if a $00 character, resetting it otherwise. Force high ASCII. Restore registers. Exit with character in accumulator and carry cleared for nonzero character.
$48A8–48CE	— **SETUP DOS TEXT FILE READ** A previous OPEN command will leave the address of the first track and sector list in the IOB. Get these addresses, check them for legal track and sector values, and load the track and sector list to the $0C00 buffer by using RWTS and the IOB. Then read the first track and sector off the TS list, storing to DOS buffer at $0D00, again using RWTS and the IOB. Set the DOS T/S pointer to the first entry on the list.
$48CF–4900	— **READ A TEXT FILE SECTOR** Increment to the next T/S pair. Abort if zero. Range check for legal track and sector, putting them into the IOB. Read text file using RWTS and IOB, storing to DOS buffer at $1D00. Then fall through to the DOS character read module.
$4901–4919	— **READ CHARACTER FROM DOS TEXT FILE** Test $F0 text file pointer. If $00, the old text file has been completely used, so go read another sector. Test keyboard, aborting on escape and clearing WPL. Read the text file character and advance the text file pointer.
$491A–492B	— **READ SECTOR VIA RWTS** Force an $01 read command into the IOB. Load registers with IOB address. Do read via RWTS using IOB values. Test carry flag. If clear, no error and the read was done into the selected buffer. If carry set, an I/O ERROR results, so vector to DOS error processing.
$492C–4950	— **PRINT DECIMAL VALUE TO SCREEN** If accumulator holds a negative value, print a minus and twos complement the number. For either sign, convert value to decimal and then print to screen. Note that the X register holds the number of digits, thus blanking any leading zeros and doing a left justify.

Table 12-7—cont. **Detailed Script of Apple Writer IIe Main Program**

$4951–496E	— **HEX-TO-DECIMAL CONVERSION** Store the hexadecimal value to the hex stash $C0,C1 with the low byte in $C0. Clear the decimal workspace $16F0–16F4 to all zeros. Start shifting and adding the hex value until equal or greater than 10. Then subtract 10 to get the decimal digit for that decade. Change this digit to ASCII by ORing with $B0. Store the decimal digits to the file, with the units at $16F0, the tens at $16F1, etc. Repeat the shift-add-test-remove-ten until nothing is left in the hex stash. Note that the X register holds the number of digits converted, thus automatically eliminating any leading zeros.
$4984–49C0	— **DECIMAL-TO-HEX CONVERSION** Empty the hexadecimal stash $C0–C1 to zero. Clear the arithmetic mode flag $A8 to $00 absolute. Read the keybuffer and test for a "+" or a "−", saving either to the mode flag. Get the first decimal digit and convert from ASCII to decimal. Multiply the old hex result by 10 and add the new decimal number to it. Repeat until you run out of decimal digits. When finished, check the mode flag. If relative minus, twos complement the hex result by subtracting it from zero. Note that a twos-complement number can be added to perform relative subtraction.
$49C1–49E7	— **HEXADECIMAL MULTIPLY BY 10** Take the value in the hexadecimal stash and double it. Then "octuple" it. Then add twice the value to eight times the value to get a 10 times result.
$49E8–49EC	— **HEXADECIMAL MULTIPLY BY TWO** Take the value in the hexadecimal stash $C0, C1 and double it by left shifting. Note that an MSB overflow cannot occur since the decimal values being converted are always less than 65,536.
$49ED–49FC	— **PRINT INVERSE TO SCREEN** Test the character. If a space or a carriage return, do not change. If anything else, force low ASCII, equivalent to the apple inverse character screen code. Put on screen. Used for status lines.
$49FD–49FE	— **CLEAR THE SCREEN** Load a form feed and fall through to print normal to screen. Form feed scrolls everything out of sight.
$49FF–4A06	— **PRINT NORMAL TO SCREEN IF NO WPL** Test the WPL flag $CF. If WPL is not active, print to screen.
$4A07–4A66	— **GET USER STRING FROM KEYBOARD** Affirm flashing cursor symbol. Get key. Disallow $00 entry and try again. If a right arrow, move to right without changing keybuffer. If a delete or a backspace, remove the previous character from the screen and back up the cursor. Continue getting characters until a carriage return. Check length and sound ding dong if >128 characters. After accepting characters, clear screen if not WPL.
$4A67–4A8F	— **DOS ACCESS PROMPTER** Unsplit the screen. If not WPL, clear the screen and put down DOS command menu. Then put down return prompt. Then get user response and process DOS command. Repeat loop until a carriage return or a DOS error.
$4A90–4AF8	— **PROCESS DOS ACCESS COMMAND** Force uppercase and save command. Abort on carriage return or anything less than an ASCII "A". If a catalog, get slot and drive and go catalog disk. Range check and exit if more than ASCII "G". If a legal command scan through the DOS selection menu until a "." marker following each letter prompt. Compare the letter before the period against the saved DOS command. If not identical, continue searching the DOS selection menu. When match is found, output the DOS prefix, then get and output command, forcing uppercase. If a slot and drive trailer is needed print a comma, followed by the trailer and a carriage return. DOS should now do its thing. If a catalog, put down a return prompt. Note that this service loop is a sub-sub. If you simply RTS, you exit to the DOS access prompter. If you pop the stack twice and exit, you cancel DOS access and return to processing words.

Org.
Let's start with the differences . . .

Apple Writer IIe Nonstock DOS

1. The DOS image is installed $3800 bytes higher in RAM than usual.
2. The boot program is OBJ.BOOT, instead of HELLO.
3. The INIT feature only inits; it does not copy a DOS image or a HELLO program.
4. Only two open files are allowed at any time.
5. The OPEN command is sometimes altered to prevent creating a new and unwanted filename.
6. DOS is accessed in nonstock ways when reading or writing a text file.

So, the DOS is stock DOS 3.3. But, then again it isn't. The important thing is that the files created are 100 percent compatible with DOS 3.3 or 3.3e.

Let's see why this DOS has to be different. It is located $3800 bytes higher than normal, so that the DOS image runs from $D500 through $D7FF, rather than from the usual $9D00 through $BFFF. All of the individual routines sit where you would expect them; just add $3800 to each address in *Beneath Apple DOS*. This higher relocation makes room for a bigger text file. This is crucial in the "E" version, and very convenient in the "F" version. When the program is first run, internal hooks are automatically connected as needed to access DOS in this nonstock location.

Since the *only* thing that ever needs booting is the boot code, this DOS has been modified to boot a program named OBJ.BOOT, rather than HELLO.

The INIT command is also modified to *only* initialize. Sadly, this does *not* give you more storage space on your program diskettes.

There is only enough room left in high RAM for two sets of DOS buffers. Thus, your MAXFILES value is always two. This is all you need for normal use anyhow.

At certain times, the DOS command of OPEN is modified to prevent creating a new file and filename. When you OPEN a file to read from it, you normally do *not* want a new file created just because of some fumble-fingered typing. This eliminates the hassle of having lots of nonexistent and confusing files mysteriously cropping up on your disks. The OPEN command is returned back to normal after special uses.

As you have discovered by now, you are pretty much "insulated" from direct DOS access by the program. The single most important and most valid reason for this is error recovery. It is essential in a word processor to be able to recover from all possible errors with your text files intact and salvagable and with your program up and running in an orderly and expected way.

There are four ways DOS is accessed . . .

DOS Access
1. LOAD does a fast and powerful custom text file load by using a special buffer in main RAM.
2. SAVE does a fast and special text file save.
3. TAB and PRT access uses DOS in the normal way. So do the [O] commands of Catalog, Lock, Rename, Verify, Unlock, Init, and Delete.
4. Update 1.1 does a stock binary load but puts the image into the LOFILE text file area.

We'll start with the normal stuff. Both the tab file and the print constant files are binary images of the work files in main RAM. These are BLOADed and BSAVEed in the usual way by feeding commands out the $FDED output hook COUT and letting DOS grab them.

All of the [O] auxiliary functions are also done in the normal way, by outputting a DOS attention prefix and the command in uppercase to the character hooks $FDED. This is totally standard and is done for Catalog, Rename, Verify, Lock, Unlock, Delete, and Init.

Remember that the [L]oad feature of Apple Writer lets you casually scan through a text file for some beginning or ending delimiters. Stock DOS does not let you do this. So, a special pair of buffers are set aside in the workfile at $0C00–0DFF. The first buffer holds a track and sector list for the newly OPENed file. The second holds a 256 character image of the last sector read. Since you have this viewable text buffer in front of you, it is a simple matter to scan for starting or ending match strings, picking up only what you need. Should you get to the end of the text buffer, the code automatically picks up the next sector to be read.

The read and write to track and sector RWTS machine language access is used to fill this buffer. The companion input-output block IOB is stashed at $02E0. Parameters in the IOB control the RWTS access.

Which lets you do mind-boggling things like counting all words in a DOS text file whose second letter is a "k". Or, more significantly, doing random access on a sequential address file made up of variable length entries.

Besides being much more powerful, this custom load method is faster than the usual DOS text file READ and WRITE commands. The OPEN and APPEND commands needed before a load or a save are done in the normal way by sending the message out the hooks as usual. This is done invisibly and automatically.

As mentioned before, the [L]oad command does not create a new text file if one cannot be found on disk. Instead, a FILE NOT FOUND error message is created. This prevents mistyping from creating nonexistent files on disk.

Loads and saves of the glossary or WPL program loads are also done using special text file access, although the fancy searching and scanning features are not normally needed.

The Update 1.1. file feature is rarely needed. When it is, a BLOAD command is sent to auxiliary DOS through the usual output hooks. This gets a binary file used by Apple Writer 1.1 and puts its image in the same place in auxiliary RAM that a normal Apple Writer IIe text file image would go.

There is no provision to save a 1.1 binary file, nor does there seem to be much demand for one. Once a text file, always a text file. Although, I could think of lots of sneaky "limit pushing" uses for this feature.

The main [S]ave command uses custom code to take a text file in auxiliary RAM and send it to DOS. This is much faster than the usual DOS access.

Summing up, there are four major ways DOS is accessed. DOS is used for the TAB and PRT file binary access, as well as for text file, glossary, or WPL loads and glossary saves. Text file loading is done via some powerful searching code that works on buffered images of the DOS sectors using RWTS and the IOB.

Text file saving is also done with special and fast code. Updating 1.1 files are done with conventional BLOADs.

DOS errors are trapped and sent to recovery code starting at $480B. The error recovery reaches back into DOS to find and then print an error message. Error processing continues by doing an orderly restart.

Note that all we say here applies only to DOS 3.3e.

"PHANTOM" DOS

The AWIIe code makes an assumption that is just plain wrong.

The code apparently assumes that high RAM and low RAM switch together between the main and the auxiliary page. In reality, high RAM only switches when page zero is switched. Since page zero is *never* switched by AWIIe, high auxiliary RAM is never accessed.

See page 27 of the extended text card supplement to verify this. Or try flipping the switches yourself. Or check the IIe *Technical Reference Manual.*

Because of this assumption, AWIIe tries to clone a copy of DOS into high auxiliary RAM. The intent was for main DOS to use files in main RAM and auxiliary DOS to use files in auxiliary RAM. This "double" DOS would seem to solve an apparently thorny memory management hassle.

The cloning attempt fails, and all that happens to the DOS image in main RAM is that it gets picked up, has the dust swept out from under it, and then gets plopped down right back where it was.

It turns out that there is a page zero flag at $7F that was intended to pick the "correct" version of DOS to use. This flag ends up picking the *same* DOS each and every time, since there is only one DOS image in the machine.

No harm is done, and the main DOS is available for all uses.

If you are using stock AWIIe, there is no reason to worry about all of this. But, if you are a hacker pushing the limits, we see that there is a big block of cloning code that can be replaced with a single RTS, freeing up some in-program space for custom mods. Better yet, with a totally "free" 16K of high RAM and a separate page zero and stack switching together, all sorts of exciting possibilities exist to integrate AWIIe with other program modules.

Enough said on DOS. Next, let's check into . . .

MONITOR ACCESS

There is essentially zero AWIIe use of the monitor . . .

Apple Writer IIe Monitor Use
There isn't any. (Or at most, darn little.)

After bootup, there is zero, repeat zero, use of any monitor routines in ROM. During bootup, a copy of only the "old" monitor area from $F800–$FFFF is cloned into main RAM at $F800–$FFFF. Then the ROM is turned off and left off.

While a few older monitor calls are used by oddball things like the first screen and the 1.1 Update code, the only monitor feature really used by this program is $FDED, the character output routine. No use is made of "fideyfoo," good old print to screen $FDF0 or any of the screen update code. Nor is any use made of GETKEY or any part of the "new" half of the monitor down at $C000–CFFF. Reset vectors are used when and if needed for reset error recovery.

The $FDED or COUT routine reads the output hooks CSWL and CSWH at $36 and $37 and then does an indirect jump there. This greatly simplifies DOS access.

FDED can point several places . . .

$FDED or COUT
Normally — (1) Points to DOS followed by a "brick wall" RTS when DOS is to receive characters. Except when it — (2) Points to special screen update code for loading only to the screen. (3) Points to a special entry code if the catalog is to be copied into the text. (4) Points to the printer during the direct keyboard to printer access option.

The usual use of $FDED is to send characters to DOS, nothing more. After DOS receives the character, it is bounced off a brick wall "RTS" to end access. There are three other local and specialized uses of $FDED. These local uses include a DOS load directly to the screen, a CATALOG command that both loads the catalog to the screen and to the text file, and finally, a special printer hook for direct keyboard to printer access.

All of the keystroke entry and screen update code is done inside Apple Writer IIe. In general, this is more powerful, faster, and manages memory better.

Which leads us to . . .

MEMORY MANAGEMENT

As we have seen, the text files are in auxiliary memory, while most everything else is in main memory. Text files are accessed by some code down on page one that does not change as the memory is switched between main RAM and auxiliary RAM. Routines in main RAM that need to do something to the text file do so via these page one access links.

Remember that the auxiliary RAM textfile is really two files. LOFILE starts at $0801 and builds up, while HIFILE starts at $BEFE and builds down. $FF markers define the beginning of LOFILE and the end of HIFILE. The "open" ends of both files face each other across all the remaining text file space. These "open" ends are marked with a $00 marker. LOFILE holds everything from the start of the message up to the current cursor position. HIFILE holds everything from just beyond the cursor to the last character in the file.

Characters are normally entered into the top of LOFILE. All of the entered characters are entered as high ASCII, but another routine carefully re-marks each end of each screen line with a low ASCII character instead.

Most of the usual routines enter characters to the top of LOFILE. Others will pass a character from LOFILE to HIFILE to back up the cursor. Yet others will pass a character from HIFILE to LOFILE to move the cursor forward. [B] and [E] are extreme examples.

There are several pointers which access the text file. These include pointers LOCURS and HICURS which point to the "open" ends of LOFILE and HIFILE.

You will also find a screen pointer that starts at a point in LOFILE equal to the top screen line, advances through LOFILE to the cursor, and then automatically switches to HIFILE to continue. The screen pointer keys on any low ASCII characters to count screen lines.

A printer pointer is used to scan through LOFILE to get characters. Since everything is moved to LOFILE before printing, no switch to HIFILE is needed by this pointer. There is also a general use pointer pair that accesses either LOFILE or HIFILE as needed.

As we have seen, there is never any switching into the monitor ROM. Those few times that monitor code is needed, it is accessed from its clone in high main RAM.

CHARACTER ENTRY

No use is made of the monitor KEYIN routine. If you tried using KEYIN with a word processor, you probably would drop keystrokes during hectic typing times.

Which is a no-no.

Instead, Apple Writer IIe uses its own internal routine to get keystrokes. This routine includes a dual key buffer. If your typing gets ahead of the processing, up to 64 keystrokes are saved in a pair of storage buffers. The main keystrokes are saved to the character buffer at $1740, while the "<open-apple>" and "<closed-apple>" keystrokes are separately saved to the apple buffer at $1AC0–1AFF. Remember that the apple keys must be saved as well as the main keystroke, or certain functions would be done wrong.

Two round-and-round pointers keep track of where you are in the key buffer. A filling pointer $F3 and an emptying pointer $F2 handle this task. During nonhectic

times, the filler and the emptier stay together and the keystrokes are immediately used. At other times, the filler gets ahead, and characters get saved to the buffer.

Routines that take lots of time automatically check the keyboard every now and then to make sure nothing gets missed. A busy signal "*" prompt appears on the normal status display during busy times.

As was mentioned earlier, characters can still get missed every now and then if a sloppy typist, a bug in the keyboard encoder, and the slower insertion mode all gang up on the key buffer. The buffer seems to be working perfectly when this happens. Its just the buffer access that fouls up the works.

Characters are normally and usually gotten directly from the keyboard during nonhectic times, and otherwise gotten out of the type-ahead buffers when things happen too fast.

There are several other character sources, besides the user.

Down on page zero is a special WPL and glossary activity flag $DF. Bit 7 or the MSB "N" slot of this flag controls WPL activity, while Bit 6 or the "V" slot controls glossary activity.

If the glossary is active, the character is gotten from the glossary file. Similarly, if WPL is active, the character is gotten from the WPL program file. Sometimes the WPL file will involve itself with its $A–$D strings. If WPL is active, and if these strings are active, then the $A–$D string becomes the source for the next character to be used. You'll find a separate string activity flag at $F6 to handle this.

There is yet another source for strings of characters. Sometimes you want to use a string already in the machine, such as the "=" filename, or something else that has been previously formatted or put together. In this case, there is a special controlling string flag $AD. If set, the old string that's usually in the key buffer at $0200 gets used one character at a time. If cleared, new characters are gotten as needed from the user, the type-ahead buffer, the glossary, WPL, or the $A–D flags.

Arrgh!

The majority of the word processor's time usually consists of patiently waiting for the user to input a new keystroke.

After a keystroke is received, regardless of its source, it is filtered for control and cursor motion commands. If a valid command is found, it is carried out. If not, the character gets entered to the top of LOFILE.

Summarizing . . .

Sources of Keystrokes

(1) Directly from the user during nonhectic times.

(2) Indirectly from the user via a type-ahead buffer when the processor gets busy.

(3) From the glossary during glossary activity.

(4) From the WPL program during active use of WPL.

(5) From the $A–$D strings if these WPL strings are active.

(6) From an old string already in the key buffer if still needed.

We have seen that there are several possible sources of keystrokes, all of which are handled internally by the code. User input is accepted either directly, or else is stashed in a pair of buffers if the processor is busy. Characters can also come from the glossary, from WPL, or from a WPL $A–$D string if the controlling flags are properly set. Sometimes, an old string will be reused, rather than getting new input.

Now for some details on the . . .

SCREEN DISPLAY

The screen display has some very sneaky and complicated code associated with it. We'll first note that you can turn the screen off and on, controlled by flag $F7. Leaving the screen off speeds up WPL considerably. Naturally, it is hard to see what you are doing when the screen is off. An "off" screen is useful to display WPL menus, prompts, and whatever.

Before a screen display is updated, any routine that messes with the text files will automatically reformat the screen lines in the text file. It does this by backing up two lines from where it is, and then automatically counting how many whole words will fit on a line. Each line stops either on a carriage return, or else when there is not enough room on the line for the next word.

At that point, a marker character, usually an $8D carriage return or an $A0 space, is changed to a low ASCII $0D or $20 and restored to the text file. All old low ASCII characters are erased from the text file. The process continues forward through the text file until a carriage return is found that is already correctly formatted.

Note that anything two lines before the current activity had to already be correct, thanks to previous reformatting. Everything beyond the next carriage return is also correct. Only the mess in the middle needs to be straightened out.

The upshot of all this is that, before a screen update, all of LOFILE and all of HIFILE have end-of-screen-line markers properly placed to end each line on a whole word.

The cursor usually stays on the middle line of the active screen. Should it overflow, everything scrolls up one line. Should it underflow, everything backs down one line. During insertions, characters get "turnstiled" as far as they have to in order to reach the next carriage return.

To update the full screen, a screen pointer pair $88,89 backs up 12 lines, which is usually 12 low ASCII characters from the top of LOFILE. Characters are copied from LOFILE and put on the screen up to the cursed location. Immediately beyond LOCURS, the pointer is moved to HICURS, and the code continues filling in characters from HIFILE until 12 more lines are completed.

The flashing you see on the cursed character is purely your imagination at work. The service routine that awaits a keystroke patiently flips the cursed character on the screen between normal and inverse text. Sometimes that character is left as inverse text. One example of this is the cursor on the nonactive side of the split screen.

Note that the large and empty "no man's land" between LOCURS and HICURS is bypassed. The highest character in LOFILE ends up at the cursed location. The very next character comes from HIFILE.

Note also that the alternate character set used has no flashing mode. High ASCII screen characters appear as normal text, while low ASCII screen characters appear as inverse text.

On a split screen, only the active half of the screen gets updated on entries and changes. The inactive half of the screen is static, remembering things the way they were.

If the wraparound flag $E1 is not active, then characters are put on the screen wall to wall without regard for word breaks. Only a 79 character line is used, since room must be left for the optional "]" carriage return display to appear in column 80.

Sometimes, a user prompt is needed at the bottom of the active screen. To do this, two lines are erased, and the prompt is placed on the bottom line. Prompts are normally read as needed out of the reference file area.

Service subs are built into the screen code for the live cursor screen motions, clear to end of line EOL, scrolling, and so on.

Another summary . . .

Screen Updates
(1) Before any screen update, low-ASCII markers are placed at the end of each screen line in the text file.
(2) Everything before the cursor on the screen comes from LOFILE as does the cursed character itself.
(3) Everything beyond the cursor on the screen comes from HIFILE.

We now know something about how DOS works, how the monitor is used, where the characters come from, how they are managed, and how the screen update works. With this background, we should be ready to survey the . . .

INDIVIDUAL CONTROL COMMANDS

Let's briefly run down the control command list, seeing roughly what each command is up to. For more detail, consult Table 12-7, or your own "torn" disassembly listing and cross-reference list.

[@] is really the DELETE key, recoded to $80 from its default value of $FF. This one unconditionally knocks out the uppermost character in LOFILE and replaces it with a $00 marker. It then backs LOCURS up one character.

[A] is not used and was probably reserved for modems and telecommunications.

[B] moves all the characters from LOFILE into HIFILE, placing the cursor to the beginning of the text. When finished, LOFILE will be completely empty, and HIFILE will hold the text being processed.

[C] changes the case flag, initally from none to U, or later from U to L, or from L to U. When characters are entered, this flag is checked. If active, uppercase or lowercase is forced as chosen. The flag is reset on all cursor motions except the left and right arrows. These arrows let you capitalize or uncapitalize as many characters in a row as you want. Only "real" letters are changed.

[D] toggles the data direction flag, between "<" and ">". This sets the direction of a search or replace. If a [W] or [X] and ">", words or paragraphs are *restored*. If "<", words or paragraphs are *deleted*.

[E] moves all the characters from HIFILE into LOFILE, placing the cursor at the end of the text. When completed, HIFILE ends up completely empty, and LOFILE holds all of the text being processed.

[F] does either a search or a search and replace. Delimiters are interpreted, substituting special ones if used. Then the text is searched using the $98,99 pointer

pair. If a replace, text is moved from HIFILE to a work buffer, and the replacement is then made. Various options substitute for fake carriage returns, allow repeats for all occurrences, use wildcards, and provide any-length capabilities.

[G] either sets up or reads the glossary. If a valid read, the glossary flag is set as a source of characters. These characters are gotten from the glossary work file until the next carriage return. At that time, the flag is cleared. If an "*", the glossary is emptied by placing a zero at the glossary start location $1B00. If a "?", the end of the glossary is found, and the new definition is entered, ending with a carriage return and a $00. The glossary has a *nest* that works like a subroutine, remembering up to eight return pointers that pick back up on the caller when the callee is finished.

[H] is the left arrow. By itself, it backs up one location by moving one character from LOFILE to HIFILE. With the "<closed-apple>" key, it does an express "by word" backspace, continually backing up until the first space is found. With the "<open-apple>" key, it saves a character to the swallow buffer instead of HIFILE, and increments the round-and-round swallow buffer pointer $AC.

[I] does an actual tab. The present position since the last carriage return is calculated. Then a test is made to see if any valid tabs exist beyond the present position. If so, spaces are added to the top of LOFILE to move to the next tab position. If the "<closed-apple>" key happens to be down, the cursor is moved without space padding. This tabs over the characters without moving them.

[J] is the down arrow. By itself, it moves characters from HIFILE to LOFILE, repeatedly frontspacing until one line is moved. Each succeeding line ends with a low ASCII marker. With the "<closed-apple>" key, it trys to go forward 12 whole lines if enough text is left.

[K] is the up arrow. By itself, it moves characters from LOFILE to HIFILE, repeatedly backspacing until one line is moved. Each preceding line ends with a low ASCII marker. With the "<closed-apple>" key, it tries to go backward 12 whole lines if enough text is available.

[L] is the load command. Loading can be from the text file, which is really a copy command, or from DOS. A fast DOS load using RWTS and an IOB is used to transfer sector images into a work file at $0D00. From this work file, delimiters can be checked, and only those needed portions get moved to the top of LOFILE, entering them into text just beyond the present cursed position. Options provide for all occurrences, wildcards, and fake carriage returns. There is also an option to load only to screen.

[M] is the carriage return that ends each command. It is not available for other uses, although you can fake a glossary carriage return with a "]", and a searching carriage return with a special delimiter, such as ">".

[O] is the DOS access menu. The menu is put down and a selection is gotten. If needed, a filename and a slot and drive is also picked up. The menu is then reread to DOS, forcing the first word of the selection to upper case with the proper prefix and postfix. For all but Catalog, the DOS code does all the work, completing the Rename, Verify, Lock, Unlock, Delete, or Init. On a Catalog, built-in code handles the prompt needed for long listings, as well as allowing a special catalog-to-file option.

[P] updates the print/program file or carries out a WPL command. On a valid two-character print/program value, that value is converted to hex and entered into the correct slot in the print/program file. Absolute values are entered as such. Relative values are added to or subtracted from the old value. Two's complementing is used for subtraction. On TL and BL entries, the string is placed in the correct file. On UT, the underline token is saved. On NP, CP, and WPL commands, the selected command is completed.

[Q] is the additional functions menu. Both tab and print/program values are BLOADED or BSAVED as called for, using $FDED standard access COUT to DOS.

Glossary loads and saves are done using stock text file DOS access. Carriage return displays or data line displays are done by toggling their respective $D2 or $E1 flags. On a [Q]-I, one character at a time is routed directly from keyboard to printer, aborting on a [Q]. To update a 1.1 file, [Q]-J reads a binary file directly into LOFILE using DOS in auxiliary RAM. The [Q]-K selection is both useless and unnecessarily nasty.

[R] toggles the replace mode flag $F5. When in replace mode, a character is deleted from HIFILE before each character entry, and then the new character is entered into the top of LOFILE as usual. The combination of deleting the character beyond the cursor and entering one at the cursor gives the illusion of replacing the old character. Replace mode is aborted on practically all cursor motions.

[S] is the save command. On a full save, the entire text is moved to LOFILE, and the text is then written directly to DOS using the RWTS access and the IOB. On a partial save, the text is scanned for the correct delimiters, and only the desired portion is saved to disk.

[T] sets or clears tabs. On a purge, the entire tab file is cleared to all zeros. On a clear, only one pair of tab entries is zeroed. On a Set, the present position is set since the last carriage return is placed in the tab file. Up to 32 tabs are allowed. A separate tab status display is updated, causing all set tabs to appear in inverse, and all cleared tabs as normal.

[U] is the right arrow, or frontspace. By itself, it goes forward one location by moving one character from HIFILE to LOFILE. With the "<closed-apple>" key, it does an express "by word" frontspace, continually going *forward* until the first space is found. With the "<open-apple>" key, it retrieves a character from the swallow buffer instead of from HIFILE, placing the character into the top of LOFILE, and decrements the round-and-round swallow buffer pointer $AC.

[V] toggles the verbatim flag $D0. With this flag set, all control characters except [M] or [V] are entered directly into the text file. This allows *imbedded* control characters for such things as special printing or typesetting commands. With the V flag cleared, control characters are used in their normal manner.

[W] inserts or deletes a whole word, depending on the data direction. On "<", a word is saved to the word and paragraph deletion buffer starting at the first open spot available. Characters are removed from the top of LOFILE and placed in this buffer, until either a space or an empty file is found. On ">", a word is recovered from the word and paragraph deletion buffer, putting the characters into the top of LOFILE, and stopping on a space. A round-and-round pointer pair $94,95 keeps track of positions in the deletion buffer. A separate deletion overload counter makes sure the buffer does not overflow.

[X] is identical to [W], except it inserts or deletes an entire paragraph, keying on a carriage return rather than a space.

[Y] is the screen splitting switch. On a [Y]-Y, the split screen is set up, using only 12 lines per display instead of the usual 24. One side of the split screen is active at a time. The other side is a static display of the way things were. Pointer $F8 decides which side is active. On a [Y] with a split screen, control flips over to the other screen side by toggling $F8. On a [Y]–N, the pointer is cleared, allowing the normal full screen display.

[Z] toggles the wraparound flag at $E1. Wraparound is always present in the text file, since each screen line ends with a low ASCII marker. If this flag is active, the screen update code ends each line on these markers. If there is no wraparound, characters are put on screen as they occur, stopping at 79 screen characters. The rightmost character slot is always reserved for a possible "]" carriage return display, whether or not it is used.

And [Z], of course, is the end of the alphabet. Unfortunately, we aren't quite through, since we have saved two of the heavies for last. As a reminder, we are

scanning through the various features of this program to see roughly what they do. Much more detail is found in Table 12-7 and in your own "torn" disassembly and cross reference.

The first heavy is . . .

PRINTING

Unlike some word processors, the Apple Writer IIe printing routines are part of the machine-resident editing code, rather than a module separately loaded off the disk.

In Apple Writer IIe, you have a choice of four possible print destinations. You can print to a real printer to get a hard copy. You can print to a modem or other special plug-in card. You can print to the screen to see exactly what your printed text will look like. Or, you can print directly to a disk text file.

This last option gives you a final document in final form, without any AWIIe imbedded commands and looking exactly like the final image to be sent to the final printer. "Printing to .pd8" is particularly useful when you are typesetting or transmitting between two different brands of computers.

The choice of a print destination is done with the ".pd" command. A ".pd0" outputs to the screen for "what you see is what you get" previews. A ".pd1" dumps to a printer card in the selected slot. Rarely, a ".pd4" could be used to dump to a modem or some other special card. A ".pd8" dumps directly to the disk.

Printing begins by moving everything to LOFILE with an [E] command. A printing pointer pair $90,91 then moves through the text file, starting at $0801 and working its way through the file.

Pages are formatted using the print/program values, such as the top margin, left margin, right margin, bottom margin, page numbers, etc. At the beginning of the first page, the ".pn" page number is saved to the running page counter pair at $BE,BF. The default left and right margins are saved as well. This way, the top and bottom line formats will stay the same throughout the document.

The top line, if used, is formatted and printed first. This is done by reading the three possible pieces out of the top line file and then moving them into a work area where the page number can be substituted for the # symbol and the final length counted. Each left, center, or right piece is then moved to a line buffer that has been previously filled to all spaces. The left piece starts at the left. The center piece starts half way across minus half the length of the center text. The right piece begins short of the right margin by its length.

After the top line, the top margin padding is put down, followed by the body of the page. The body is formatted and printed one line at a time, allowing for paragraph margins or outdents on the first line in each paragraph.

Each line begins by getting enough characters out of the text file to fill the line. As the characters come in, they are filtered for AWIIe imbedded commands and for footnotes. Imbedded commands start with a carriage return, followed by a period, followed by two or more letters. If these commands are found, the printing stops long enough to allow the imbedded command to do its thing.

For instance, on a ".1m+5" command, printing halts momentarily. The left margin is then gotten, decimal 5 is added to it, and left margin is then replaced. The new left margin value will get picked up on the next line.

Unfortunately, as you undoubtedly have found out by now, AWIIe counts any command it does not recognize as printing characters. Which leads to the "short line"

problem on certain printers that receive imbedded escape commands. A short line patch is available free; just call me on the helpline for a copy. It also appears on the companion diskette.

Characters are also filtered for footnotes, which begin with the "(<" command. If footnotes are found, they are stored into the footnote buffer at $1200, and the footnote flag $FE is set. This flag is incremented once for each footnote per page as needed. The very first footnote knocks two counts off the available number of printed lines, while any additional footnotes knock off one extra line. This gives a space between the bottom body line and the first footnote line.

At any rate, characters are gotten and filtered until there are enough whole words to fit between the left and right margins. These are placed in the line justify buffer at $1600. Then that line is justified.

Should left justify be in use, nothing more is done, leaving all of the words flush left. If center justify is in use, the length of the characters used are subtracted from the line width, and half this distance is used to offset the characters in the line buffer.

If right justify is in use, the length of the character string is subtracted from the line width, and this entire distance is used to offset the characters in the line buffer. In any of these three modes, you end up with the buffer holding the line justified in the correct position.

Spaces are added as needed before the center justified and right justified text. Spaces are not needed *beyond* any text since the carriage return completes the entry. A row of printed spaces looks the same as the unprinted page, so it is neither needed nor used.

On fill justify of a long line, the needed number of padding spaces is calculated. Text is then moved one space to the right, beginning with the first space, and repeating as often as needed to force the fill justification. Microjustification is not available inside stock AWIIe. You can instead use imbedded commands to tell an intelligent printer to do this for you. Naturally, if you have this feature available, it will look much better than text justified by whole spaces.

The *Diablo 630* is especially good at microjustifying AWIIe text, eliminating the short line problem at the same time. More details on fully automatic formatting for high print quality appear in Enhancement 9.

Regardless of the justify mode, all of the characters end up in the correct place in the line justify buffer. When finished, this line is output for printing. The line is preceded by enough spaces to make up the left margin. On first paragraph lines, the ".pm" value is also used to adjust the number of leading spaces needed.

As the line is printed, the characters are filtered for the underline token. Should this token show up, it is replaced with a space, and the underline mode flag $E0 is toggled. Underlining is done by printing the "_" underline character, and then by backing up one space and printing the character to be underlined.

Underlining will not work on some very old or otherwise primitive dot-matrix printers. The printer must be able to recognize the $88 ASCII backspace command for this type of underlining.

Incidentally, one simple AWIIe fix for underlining up to a period or comma is to imbed a backspace before the punctuation. Most often, your printer will offer better underlining than AWIIe does.

As many lines as are needed are put in the body of the text. When finished, any footnotes are recovered from the footnote buffer and printed. These are followed by the bottom line padding, and, if used, the bottom line.

Printing continues until all of LOFILE has been printed. At that point, a new file can be loaded and a ".cp" continue printing command can be given, picking up exactly where you left off with the same running page number and current margin settings.

On the single sheet option, printing halts at the bottom of the page long enough for you to change papers.

By the way, your printer card should defeat any video echo. If it does not, this may slow down your printing, particularly at higher serial baud rates. You will, of course, get the best printing with an intelligent printer or typesetter that accepts imbedded commands and can do its own proportional spacing, boldface, italics, shadow printing, and microjustification.

So much for printing. The real biggie is . . .

WPL

The amazing thing about WPL is how much is done with how little. The additional code needed is rather short and compact.

WPL is a supervisory language that looks like a cross between Pascal and assembler. Its intended use is as an executive controller that will see you through long and involved tasks.

Obvious uses are printing a multiple file book chapter with the correct headings and footings, customizing a mailing to a separate address list, counting words, putting down menus, prompting operators, building an index, and things like this.

It's the nonobvious uses of WPL that boggle the mind.

I've used WPL to insert or remove the line numbers from assembly code, and to *picture process* strings sent to a plotter. I've used it to trick a printer into doing "camera ready" copy, and to scan for formatting errors. I have also used it to create high-level graphic images. In fact, I am convinced that WPL is far more powerful at processing pictures than it is words. Others have even written adventures in WPL!

Back in Enhancement 9, we saw how to use WPL to completely format a document for full "bells and whistles" high quality printing, along with several other examples.

The message is overwhelming. WPL is super powerful and super important. Without it, Apple Writer IIe may have some second-rate competition.

With WPL, that's all she wrote . . .

> If you do not thoroughly know and aggressively use WPL, then you are passing up at least 98 percent of the good stuff you can do with AWIIe.

Maybe more. So get with it. Now.

Fig. 12-4 summarizes a WPL instruction. WPL instructions are each one line long, ending with a carriage return. Lines are normally done in the order they are found in a WPL program, although there are several important exceptions.

Each line may begin with a label. The label must not have any spaces in it. If used, a label lets WPL find a certain line for possible subroutine or jump access. If, as is more often the case, a label is not used, then a space must be the first character on a WPL line.

Either way, the first character after the first space in a WPL line is treated as if it were a control character. WPL then behaves just like you typed that control character from the keyboard. For instance, if a WPL line consists of a space followed by a "B", the WPL interpreter would see this as a [B] and would move the cursor to the beginning of the screen.

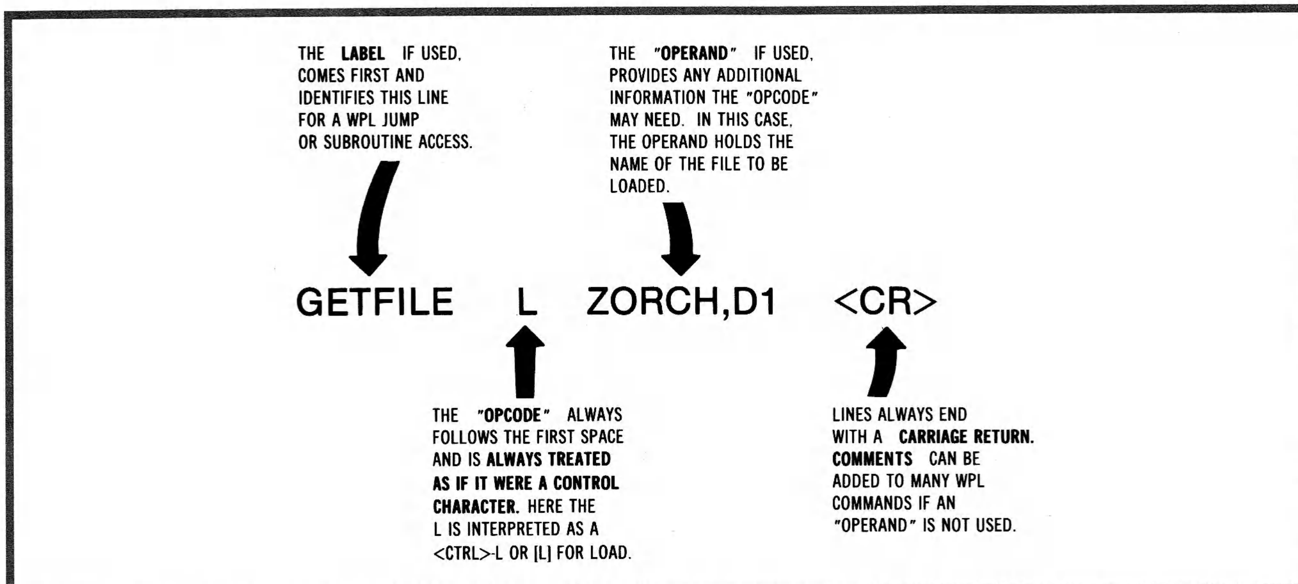


FIGURE 12-4. A WPL command line is very similar to a line of assembly source code.

Nearly anything you can do at the keyboard, WPL can do for you, automatically, potently, and error free.

Think of WPL as a high-level language that is extremely proficient at editing long strings of characters and acting on them, as well as being a competent disk and printer supervisor.

So, what is WPL and how does it work?

We first need a way to write a WPL program. Since a WPL program is nothing but some processed words, you write your WPL program on Apple Writer IIe just like any old text file and save it to disk.

There is a WPL command called DO. To *run* your WPL program ZORCH, you simply do a "[P] DO ZORCH."

That is all there is to it. The DO code first clears all the various WPL flags and work areas. It then loads the named program into a WPL program file starting at $0E00. The program can be 1024 characters long with footnotes in use or 2048 characters without. WPL programs can be chained together. Variables are preserved during this chaining. The DO command also sets the WPL activity flag $DF, so that keystrokes will be read from the WPL file, rather than from the keyboard.

If the WPL flag is set, the first line of the WPL program is read. If a label is present, it is passed over, finding the first character past the first space or string of spaces. This character is converted into a control command, and is processed just the way any control characters entered from the keyboard would be. Any remaining characters on the line are used as needed by the control command. For instance, a filename might follow a "L" for [L]oad, or a search and replace string might follow a "F" for [F]ind.

The WPL lines are read one at a time, usually in sequential order. Each line terminates with a carriage return. The final WPL line ends with a $00 marker, which stops WPL and returns control back to the keyboard.

WPL has jumps and subroutines. The WPL command GO will start at the beginning of the WPL file and search for a label. On the "pgo" jump command, the label is found, the program unconditionally jumps to that line and then continues from there.

The WPL command "psr", for subroutine, does almost the same thing, except that a return address is remembered on a WPL stack at $1700, along with a stack pointer $92 that remembers where to return to.

WPL has numeric variables. Not very many, though. There are three numeric variables (X), (Y), and (Z) that each can range from 0 to 65535. Any time an (X) is found, the (X) value will be substituted. Same goes for (Y) or (Z). You can set these numerics to any value, either absolute or relative. You can easily test a numeric for zero. With some hassle, you can also test a numeric for most any nonzero value.

For instance, "psx45" puts a decimal 45 into (x). "psx7" sets (X) unconditionally to decimal seven. A command of "psx+7" adds seven to whatever was already in (X). Most importantly, the command "psx-1" decrements a counter loop involving (X) by a single count.

The numerics are really nothing but print/program values, and are stashed in the print/program file just like, say ".lm" or ".ut". See Table 12-1 for the exact locations. Substitutions are done at the time the WPL line is interpreted.

WPL has string variables. There are four of them named $A through $D. They are stashed in the work files starting at $1780. Just like the numerics, the strings are substituted for their symbols at the time the WPL is interpreted. Strings may be loaded from memory or disk with the ".ls", assigned to an immediate value with ".as" and compared with the ".cs" commands. Check Tables 12-5 and 12-7 for more details.

During disk access, the ".ls" load string command borrows an unused portion of the text file immediately *above* LOCURS out in "no man's land." Since the $A–D strings are only allowed to be 64 characters long, there is little danger of crashing into HICURS, except on a nearly full text file. This is done to give all of the powerful loading options to WPL strings that the usual text loads receive. After use, the string above LOCURS is zeroed out so it does not become part of the text file.

WPL has conditional execution, an absolutely essential feature of any computer language. The next WPL statement gets skipped if a numeric reaches zero; or if [F]ind can't; if [L]oad won't; or if "psc" doesn't compare. The skipped statement usually is a jump, a subroutine call, or a program quit. Thus, you can make a test and cause WPL to pick two different routes, depending on the result of that test.

WPL interacts with the user. You can clear or put fixed messages or menus on the screen with the "ppr" command. You can get a string from the user with a "pin" command. The display can be turned on with the YD command and off with the ND command. An off display computes much faster, besides holding the last prompt or message for you.

There is also an ".ep" command in WPL that enables the printer if the hex stash value is not zero. You use this command to print only the page you want in the middle of a document. To accomplish this, put an ".ep0" at the beginning of your document, and an ".ep1" where you want the actual printing to start.

Let's round up these . . .

WPL Tricks that Beginners Miss

Clear screen	ppr[L]
Beep	ppr[G]
Tweedle	ppr[G][G][G]
Turn printer off	.ep0
Turn printer on	.ep1

Clear screen only works after a "pnd" command. It is best to use the beep only when an error occurs. Use the tweedle to get the user's attention when a long routine is finished.

Of course, WPL can have errors. Lots of different things can go wrong with a WPL program. You might have a label missing or mispelled. You might be calling subroutines without returning. The program might get too long. Just as DOS has an orderly exit method and prompting for DOS errors, there is a separate WPL error processor that does an orderly shutdown of WPL and prompts the user. This error processor starts at $3B45 in the "F" version.

The escape key shuts down a wayward WPL program. This same key also serves as a printer panic button.

WPL is great for what it is and what it does. But it is a specialized language and, as such, it has some series shortcomings. Its arithmetic abilities are putrid. A floating point multiply takes over 46 seconds. That's seconds, not milliseconds or microseconds.

So, there are lots of things you might like to add to WPL to make it even more powerful.

Of the things I would like, the ability to PEEK, POKE, and CALL machine language extensions is essential. Also needed is a single keystroke GETKEY from the user, and most importantly, a way to "put the cursed character into the $D string."

We'll find out how to add a surprisingly simple POKE capability to WPL shortly.

Which leads us to the obvious question of . . .

MODIFYING APPLE WRITER IIe

Obviously, you want to modify either all of Apple Writer, or else just part of WPL, or you wouldn't have gotten this far.

First, you will want to have completely torn apart the program and thoroughly understand it.

Secondly, you will want to find some good . . .

> Hooks —
>
> Places in a program where it is reasonable to attach your own custom code.

A listing of a few of the more obvious attachment and extension points of the "F" version of Apple Writer IIe is shown in Table 12-8.

As you can see, there are control functions you can grab, auxiliary functions you can divert or add, blocks of memory you can replace with your own code, and large portions of RAM that are unused.

Your first attempt at any modification should preserve the position and length of the stock code . . .

> If you modify Apple Writer IIe, try to keep the position and the length of each code module EXACTLY the same as the original program.

Table 12-8. Apple Writer IIe Hooks

Hooks are points in a program that lend themselves to your own custom expansion and modification.

Here are some reasonable "F" version possibilities:

1. Control functions []], [/], [|] and [^] are addable for command modes. Function [A] is also usable but may conflict with modem use.
2. Auxiliary functions [Q]-H, [Q]-J, and [Q]-K are easily diverted. [Q]-H does the same thing as the <esc> key. [Q]-J might be usurped, since just about all 1.1 files should be long gone. Selection [Q]-K destroys everything within a 300-foot radius of your Apple IIe and then "kicks sand in your face." It is even worse than a cold restart.
3. Auxiliary functions [Q]-L through [Q]-Z may be added. For that matter, you can do a [Q] followed by a number or punctuation key for even more tasks.
4. The following internal code areas can possibly be diverted for your own uses:

(a)	Quit Apple Writer	$2CA4–2D0E	(106 bytes)
(b)	Convert 1.1 files	$2D0F–2DB1	(162 bytes)
(c)	DOS cloner	$3464–3480	(28 bytes)
(d)	Reserved area	$3AD6–3B39	(99 bytes)
(e)	Reference file	$5088–508B	(4 bytes)

Note that (a) and (b) are contiguous.

5. The top of the print/program file from $19E2–19EF is available for values you wish to save to disk as a PRT value.
6. All of main RAM from $53D0–BFFF is available free for your use, although future-looking uses should avoid $BF00–BFFF. Page zero locations unused by Apple Writer IIe, DOS,or actually used monitor routines are available, as are main RAM locations $0157–0180, $0380–03CF, and $16E5–16FF.
7. Also available, but tricky to manipulate, are auxiliary page zero and page one, auxiliary memory high RAM, and the two "first" 4K banks of RAM in both main and auxiliary memory.

To not do this will introduce all sorts of sticky complications.

Which says that you should hold off trying to use the HIRES screens *in their intended locations* for anything unless you are ready to change things in a very big way.

There are some surprisingly simple ways to do HIRES screen dumps on either stock or modified versions of AWIIe. Much more on this and an AWIIe graphics in general in a future enhancement.

Let's suppose that you want to make some small and sane change to Apple Writer IIe. How do you do it?

I'll show you two wildly different ways to modify the program. The first method is shown you in Table 12-9. This method is best suited for something you wish to sell. It takes a stock copy of Apple Writer IIe, installs it in your machine, and then overwrites your patches into portions of the machine-resident code.

Most importantly, it keeps separate the original work by the original author and your own. If your patch is good, the original author sells more copies of his work. If your patch is bad, you get blamed or ignored.

You make your patch by BLOADing the original program into your machine under stock DOS 3.3, changing what you want to change, and then saving the program under the name PATCH.

The patch is installed by a sneaky trick. Note that the print/program file sits *before* the main program in memory. Normally, when a print/program file is loaded, it fits exactly into the print/program space. But, thankfully, there is no check made on the *length* of a print/program file as it is being BLOADed.

Table 12-9. How to Patch Apple Writer IIe

WARNING: The following description works ONLY on EXACTLY the DOS 3.3e "F" version of Apple Writer IIe, circa March of 1983. Use the "tearing method" to change as needed to suit your current code version.

TO PATCH APPLE WRITER IIe:

(1) Write protect your third backup copy of Apple Writer IIe and your third backup copy of your DOS 3.3e system master.

(2) Cold boot your DOS 3.3e system master disk third copy in slot 6, drive 1.

(3) Remove the system master disk from drive 1 and insert a new and blank diskette in its place. Do not use drive 2.

(4) Init this new, blank diskette with your favorite HELLO or MENU program. If you have no favorite HELLO program, then . . .

] 10 REM TEMPORARY HELLO <cr>

] INIT HELLO <cr>

Do NOT use the init feature built into your Apple Writer program! Use your system master.

(5) Replace the blank diskette with your third backup copy of Apple Writer IIe in drive 1.

Then . . .

] BLOAD OBJ.APWRT] [F, A$2300 <cr>

] BLOAD PRT.SYS, A$18C0 <cr>

] POKE 6912,0 <cr>

(6) Replace the copy of Apple Writer IIe with the initialized blank diskette in drive 1.

(7) Make any and all changes to the code now resident in your IIe that you care to.

(8) Then . . .

] BSAVE PRT.PATCH, A$18C0, L$3B41 <cr>

Do this to your initialized blank diskette in drive one. Your particular "L$" value will be different if you lengthened or shortened the original code. This completes creation of your custom patch.

TO USE YOUR PATCHED VERSION OF APPLE WRITER IIe:

(1) Boot your stock Apple Writer IIe diskette in the usual way.

(2) Replace the stock diskette with the modified diskette.

(3) Then . . .

[Q]-C PATCH <cr>

Your patched code is now resident in your Apple IIe awaiting your use.

So, you simply load a print/program file under [Q]-C that is a little longer than normal. In fact, you load one that is so long that it *completely overwrites* all of stock AWIIe!

To use your patch, you boot *stock* Apple Writer IIe in the normal way. Then you [Q]-C PATCH, and all your changes get magically made to your machine resident program.

Two gotchas.

First, if there is any change to stock Apple Writer IIe at all, your patch may not work. Thus, it is up to you to keep track of version changes through time. Since this is such a popular and well-supported program, rare and major updates are more reasonably expected than continual small changes. Versions are obviously tested by looking at certain bytes in the code that are different for each version. Note particularly that

separate patches are needed for the "E" and "F" versions of the code, depending on whether extended memory is available.

Secondly, the length of your program may be different from the original. This is allowed, since there is some 27K of free main RAM above the present "F" program that you are free to use. Be sure to change the "L$" length in step eight of Table 12-9 if your length differs from the original.

Once again, it is best to leave everything exactly where it is in memory, making changes that overwrite, rather than relocate, code modules. If you don't have enough room, jump out into "free" RAM and then jump back.

This patching method is preferred for commercial uses. You might like something simpler and more flexible for your own personal needs.

So, let's see how we can go about . . .

ADDING POKE TO WPL

Any computer language faces a dilemma.

If you do not provide ways to extend and change the language, then your language is forever yours and its integrity is never compromised by dumb or otherwise stupid things others have tried to do to it.

If you do provide ways to extend or change the language, then the language can push the limits and do fantastic and unexpected things, as many different people put their skills and thought processes to work improving and upgrading.

The same is true of a word processing program.

If you change it, it may get better or it may get worse. If you cannot casually change the program, it does exactly what is expected of it, reliably and certainly. If you do change the program, it may do great and wonderful things. Then again, it might blow up in your face.

What I am saying is . . .

> POKE is a deadly weapon!

I will show you a very simple way to add a rudimentary POKE capability to Apple Writer IIe that will let you modify any part of the program at any time for any purpose. But bear in mind that you have been handed a deadly weapon that, more likely than not, will get you into very deep trouble.

Needless to say, neither Sams nor I will clean up any of the mess you make.

The idea is simple. Nobody uses the [O]-C "Verify File" option much. Just *replace* it with an [O]-C "Bload Patch", and you can wreck any havoc of your choosing.

One possible use of a POKE ability is to send commands to a printer card that needs POKEs to change its printing modes. To POKE with a BLOAD, you simply load a binary file one byte long. That byte goes where you tell it to. To do fancier DOS stuff, just use your new BLOAD to overwrite the normal tab loads and stores with any fancier DOS commands you may need. Just be sure to put everything back the way it belongs when you finish. You can even get verify back temporarily, again by overwriting the tab load or store.

As an "exercise for the student", try replacing the [Q]-K option with code that "puts the cursed character into the $D string." Note that you have to use the memory

management routines and that the cursor must be decremented, used, and then incremented back to point to the open end of LOFILE. The solution is included as a bonus module on the companion diskette.

Putting the cursed character into the $D string ends the long WPL song and dance needed to find out what the cursor is looking at. Many file-modifying programs can be dramatically sped up with this new feature. Additionally, logic can be done based on what any particular character is or is not.

The professional way to add a POKE capability is by following Table 12-9. Here is a simpler and more flexible method . . .

Adding POKE to Apple Writer IIe

Preferred commercial method —

Use Table 12-7, changing the [O]-C selection to read "Bload Patch" instead of "Verify File".

Alternate personal method —

(1) Make a fourth backup copy of Apple Writer IIe, using any of the usual methods. Put a fluorescent label on this and then stripe the label.

(2) Using a disk zapper, such as COPY II PLUS, find and blast out the "Verify File" in the DOS options menu and replace it with "Bload Patch," directly on this newly created disk.

(3) The needed code area may or may not be on the boundary between track 8, sector 0 and track 9, sector B.

"Verify File" codes as $D6-E5-F2-E9-E6-F9-A0-C6-E9-EC-E5-8D.

"Bload Patch" codes as $C2-EC-EF-E1-E4-A0-D0-E1-F4-E3-E8-8D.

Don't forget that there are *two* versions of AWIIe on your source disk. There's the "E" version for use without extended memory, and the "F" version with. Be sure your patch goes on the right version, or you will end up in deep trouble.

Remember that most POKEs will work on one version and will destroy the other.

Once you start using your new POKE ability to in-place modify your word processor, you will be amazed at the many different things you can do with it. Also astounding will be the disasters you create along the way. Be sure to use the hotline to keep us informed on your successes.

If any.

Let's wrap things up by . . .

Capturing Your Own Source Code

Needless to say, it is far easier to make major changes in any program by working directly with source code, rather than puzzling over mysterious object code. Captured source code is almost essential for such major changes as repositioning the program or otherwise making major improvements. It is also necessary if you are to integrate your word processor with other program modules, such as a spreadsheet, data base manager, or telecommunications package.

There are also many good reasons to *not* try and capture source code. For one thing, a lot of time, patience, and effort are involved if you are to do the job correctly. More importantly, things get sticky fast if you try to make any commercial use of your captured and modified source code.

You are, of course, free to make any changes you want any way you want to any program you personally own, so long as you do so for your own use only. The problem comes up if you try to sell or otherwise pass on "your" work, which is really a mix of what you have done and the value created by the original author.

Play fair, or don't play at all.

At any rate, Table 12-10 shows you how to capture source code for the "F" version of Apple Writer IIe. Note that these instructions will ONLY work on the *exact* version of the program that I used.

There is a slight amount of black magic involved in getting a good capture. This is handled by a module called SNEAKYSTUFF. This is listed in Table 12-10 and provided ready for your use on the companion diskette. We'll save details on SNEAKYSTUFF for you to puzzle over.

The SNEAKYSTUFF module is disgustingly elegant. I'll give a free Sams book to the first ten Apple Writer IIe enhancers that correctly tell me what this module does and what specific problem it solves.

We may take a further look at source code capturing techniques in a later enhancement.

A most important capture rule . . .

> Captured source code is totally and utterly useless if it does not EXACTLY reassemble into code that is a PERFECT match to the original object code.

So, be sure to reassemble your captured source code back into object code and byte-for-byte compare it against the original. Full details on working with source and object code appear in my *Assembly Cookbook for the Apple II/IIe* (Sams Cat. No. 22331).

See you there.

Your captured source code consists of some 7837 lines of undocumented code. This can be assembled on stock EDASM, but that's about all. From here, you will want to split the code up into workable chunks and add the proper chaining and documentation. After that you will be well on your way to making major modifications of your own choosing.

Note that this enhancement is intended specifically for Apple Writer IIe. The helpline is specifically set up to handle Apple Writer problems and has many free Apple Writer patches available. Write or call.

Table 12-10. How to Capture Apple Writer IIe Source Code

WARNING: The following description works ONLY on EXACTLY the DOS 3.3e "F" version of Apple Writer IIe, circa March of 1983. Use the "tearing method" to change as needed to suit your current code version.

(1) Write protect your third backup copies of Apple Writer IIe, the DOS 3.3e system master, and Disasm 2.2e. Work only with drive 1 in the following steps to avoid any possible mixups.

(2) Init a new blank diskette using your backup DOS 3.3e system master, as you did in Table 9. Do NOT use the init feature of the Apple Writer program.

(3) Cold boot the newly inited blank diskette. Then:

```
[ CALL -151  <cr>

* 2000: 34 09 C1 C1 00 00 00 00  <cr>
* 2008: 34 0B C1 C1 AB B2 00 00  <cr>
* 2010: 34 10 C1 C1 AB B7 00 00  <cr>
* 2018: 34 17 C2 C2 00 00 00 00  <cr>

* 2020: 34 19 C2 C2 AB B2 00 00  <cr>
* 2028: 48 02 C3 C3 00 00 00 00  <cr>
* 2030: 48 04 C3 C3 AB B2 00 00  <cr>
* 2038: 48 44 C4 C4 00 00 00 00  <cr>

* 2040: 48 4B C4 C4 AB B7 00 00  <cr>
* 2048: 48 48 C4 C4 AB B4 00 00  <cr>
* 2050: 48 0B C5 C5 00 00 00 00  <cr>
* 2058: 48 12 C5 C5 AB B7 00 00  <cr>

* 2060: 28 3E C6 C6 00 00 00 00  <cr>
* 2068: 28 3F C6 C6 AB B1 00 00  <cr>
* 2070: 28 40 C6 C6 AB B2 00 00  <cr>
* 2078: 2B 6B C7 C7 00 00 00 00  <cr>

* 2080: 2B 6C C7 C7 AB B1 00 00  <cr>
* 2088: 2B 6D C7 C7 AB B2 00 00  <cr>
* 2090: FF FF FF FF FF FF FF FF  <cr>
```

Check your work with a . . .

*2000.2097 <cr>

And then . . .

*3D0G <cr>

] BSAVE SNEAKYSTUFF, A$2000, L$98 <cr>

This creates a special label table for later use by Disasm IIe. A ready-to-use copy of SNEAKYSTUFF is included on the Enhance II companion diskette. Note that this table MUST be used with Disasm IIe. See text for address.

(4) Cold boot your backup copy of Disasm 2.2e. Then . . .

] BLOAD DISASM <cr>

Now, put the Apple Writer IIe backup disk into drive one and . . .

] BLOAD OBJ.APWRT] [F, A$3300 <cr>

Table 12-10—cont. **How to Capture Apple Writer IIe Source Code**

Next, put the recently inited diskette into drive one and . . .

] BLOAD SNEAKYSTUFF <cr>

] CALL 2048 <cr>

You are now ready to start the capture process.

(5) Complete the following prompts as they come up . . .

Assembler: DOS Toolkit (1) <cr>

Physical address beginning: $3300 <cr>
Physical address ending: $63D0 <cr>
Execution address beginning: $2300 <cr>

Table 01:	Begin $3B92	End $3BA2	F = 11 <cr>
Table 02:	Begin $4046	End $4095	F = 11 <cr>
Table 03:	Begin $5AF9	End $63D0	F = 11 <cr>
Table 04:	<cr>		

Label table address: $2000 <cr>

Wait . . . and then . . .

Printer slot:	1	<cr>
Single X-Ref:		<cr>
Full X-Ref:	N	(unless you want one now)
Generate file:	Y	(do NOT use <cr>)
Filename:	APPLEWRITER.F.SOURCE <cr>	

Wait lots more to complete the capture.

(6) Assemble your new source code using EDASM off the DOS toolkit. You MUST get ZERO assembly errors. When you finally do, compare the new object code against the original by loading both into different parts of IIe main memory and then using the monitor verify command. Like so . . .

Cold boot the diskette with your new object code on it. Then . . .

] BLOAD APPLEWRITER.F, $A6300 <cr>

Swap to the stock Apple Writer disk and . . .

] BLOAD OBJ.APWRT] [F, A$2300 <cr>

Remove this diskette. Then . . .

] CALL –151 <cr>

*62FF<22FF.53D1V <cr>

If the verify is good, you will get two and only two errors. These errors will lie outside the program boundaries at $22FF and $53D1.

YOUR NEW OBJECT CODE MUST EXACTLY MATCH THE ORIGINAL.

If it does not, find out why and try again. Be sure to delete all inaccurate capture attempts.

(7) Your 7837 captured source code lines are undocumented and far too long to do anything with except direct assembly. To continue, you will want to break your captured source code up into reasonable-sized chunks, say 16K or so for "new way" editing, and then use the CHN command to chain the pieces together. Then make up reasonable label names and insert the documentation from Tables 12-1–12-7 where and as needed. You also have the option of pre-capturing labels using a new label table and another pass through Disasm IIe.

NOTE: Full details on EDASM and the fast "new way" editing process appear in Don Lancaster's *Assembly Cookbook for the Apple II/IIe*, Sams Cat. No. 22331, from your favorite book rack, or call 1-800-428-SAMS.

The magic modules SNEAKYSTUFF and CURSDSTRING are both included on the companion diskette to this volume.

In addition, the entire contents of this enhancement are available on eight diskette sides as the AWIIe TOOLKIT.

See the response cards in back for full details.

This enhancement works on most "real" Apples including the IIc. Changes are needed on Franklins or clones.

Enhancement 13

THE VAPORLOCK

An upgrade of Enhancement 4 that gives you a fast and exact lock to video timing without needing any hardware modifications. It works on the II, II+, IIc, or IIe. Among other uses, you can now mix and match HIRES, LORES, or text nearly anywhere on the screen.

THE VAPORLOCK

The exact field sync of Enhancement 4 sure separated the sheep from the goats.

The goats said "Oh Boy! Now we can push the Apple limits and do the impossible!" And they went out and did just that. But countless sheep instead said "You want *me* to *open* the *lid* on my Apple? No way!"

Needless to say, I am a Capricorn. Like most goats, I don't even know where the lid to my Apple is. Also needless to say, should any Apple ever get caught revealing IT messages to a sheep, that Apple will immediately get busted to a rank of RAM 4K and will get reassigned to handling the data base for the latrine orderly on the northernmost military post in the world.

Well sheep, this one is for you.

The Vaporlock is a sneaky way to do a fast and exact field sync that takes *zero* hardware modifications, and works with most any "real" Apple II, II+, IIc, or IIe. It does everything the old field sync does, except that it locks far faster, requires no hardware mods, and is much easier to program and use.

The Vaporlock can be used in your commercial programs with suitable credit. Locking time can be as few as eight horizontal scan lines, and a substantial amount of "free use" throughput remains available to you during a mixed field display. Program timing is now far simpler and less critical.

If you've come in late, any exact field synchronization scheme on the Apple lets you mix and match HIRES, LORES, and text anywhere on the screen in nearly any combination, can give you glitchless and flawless animation, can greatly simplify light pen hardware, and does lots more.

We'll look at some mind-bending additional uses for exact field sync at the end of this enhancement.

RUNNING ON "FUMES"

Let's review. The Apple shares its main memory between the video display electronics and the microcomputer itself. It does this by switching between the two. The switching is done twice every CPU cycle, or roughly once each half microsecond.

This invisible memory sharing buys you transparency. Transparency is the ability to continuously and smoothly display stuff even while the computer side of the works is changing or updating the same memory area in use by the video.

In any stock Apple, there is no built-in way for the CPU to tell *exactly* where the display happens to be in its scanning process. If the CPU could find out when the display is black during its long vertical retrace, you can clean up most animation. You do this by replotting to screen memory only during the times when the display scanning won't mix up "old" and "new" data bytes. Proper use of blanking times can eliminate any "sugar" or "collisions" on the screen and make things much more viewable.

Better yet, if you and the CPU can find out *precisely* and *exactly* when a new video field is going to start, you can flip the display soft switches on the fly, and mix HIRES, LORES, and text most anywhere you want in most any order. If you do it just right, this switch flipping turns out glitchless and completely free from any display flicker. Lots of other exciting things start happening when you begin flipping nonobvious soft switches in exact screen display positions.

Thus, you can make some Apple display improvements just by finding the vertical blanking time and using it. But you can do far more and do it far better if you are able to provide an exact and jitter-free lock to the video display timing. You have an exact lock when you know *exactly* where the scan is at all times.

Exact, precise, and jitter-free field sync is not trivial. If it was, you would have known about it in 1977, for everything needed for it was available way back then, just as it is today.

There are only four ways I now know about that can be used to route information from the video side of the Apple electronics to the CPU side. Let's pick up on these in historical order.

The first way was to add the wire of Enhancement 4. As a jittery lock, this added connection dates back to the days of the *Apple Software Bank*. Back then, the average cost of an Apple program was around $11. Fully unlocked and unprotected, of course. If you couldn't afford such horrendous expense, your Apple dealer would freely and generously copy lots of zero-cost Apple-supplied public domain programs for you, customer or not.

Sigh.

Much more recently, Enhancement 4 showed you how to get from the jittery lock to an exact lock on older Apples.

This wire connected a vertical blanking signal from the video side to a port on the CPU side, which let you immediately find the blanking time with a simple software test. Given a long enough "song-and-dance" routine in software, you could also do an exact lock. The exact lock could take as many as seven full fields.

Our first locking method has several disadvantages. The Apple had to have its hardware modified. Commercial use would be severely limited, since you might have to provide the parts for the hardware mod as well as the software in your customer package. You had to tie up the cassette input, or the phantom "fourth" push button. Worst of all, there is no place to put the wire on an Apple IIc or IIe. And, finally, an exact lock took so long that there was no way to do a lock on every field.

The only nice thing you can say about this method is that it proved that an exact lock was possible and that mind-blowing things could be done with it.

The second method only briefly saw the light of day. By adding two cheap gates to an Apple II or II+, you could extract a timing signal on the last scan line of each display. The circuit tapped some of the LS161's in the main timing chain. The output of this synchronizing circuit went the same place as the wire of method one.

This new locking circuit output a burst of eight pulses. Each pulse was four microseconds wide and the spacing between pulses was also four microseconds. The pulse burst appears only on the last blank display line.

Software would read this signal and could give you an exact lock each and every field. This method greatly eased software problems and opened up lots of new possible exact locking uses. The problems here included even more hardware than a simple wire, operation only on the II or II+, and a conflict with a few commercial plug-in cards that needed access to the same timing area on the main board.

More details on this method appeared in Volume VII of the West Coast Computer Faire proceedings.

Enter the Apple IIe for the third locking method.

The IIe has a nice feedback wire built in that goes to previously unused address location $C019. No hardware mods are needed. By reading this location, you could tell whether you were in the blanking or the live portion of a scan. This location proved a godsend for programmers who would be satisfied with a jittery, nonexact field sync.

A single measurement of this location could have as much as seven characters worth of jitter. An exact lock can be done, but it takes up to seven fields of exact timing. At the very minimum, two whole fields seem to be needed for an exact lock. That takes sneaky coding. Besides, whatever you did would not be downward compatible with the II or II+. But at least there is no need for hardware mods.

The update section on Second Edition printings of *Enhancing Your Apple II,* Vol. 1 (Sams Cat. No. 21822) gives you complete details on how to use $C019 for an exact lock on your Apple IIe.

Unfortunately, the IIc uses $C019 for a wildly different purpose, and IIe routines will not run as is.

The fourth locking method is not mine. It was discovered by Bob Bishop and published in *Softalk* a while back. But, amazingly, Bob presented everyone with the keys to the kingdom and then never bothered opening the door! He showed only an approximate, jittery lock with on-screen glitches. Which did neat things. But it was not an exact lock, and thus it severely limited the really great stuff that was possible.

So, for method four, we'll be combining my proving that an exact lock can be done with Bob's paving the way for it to be done.

Method four literally runs on fumes.

Which is why I call it the vaporlock. Fig. 13-1 shows us details.

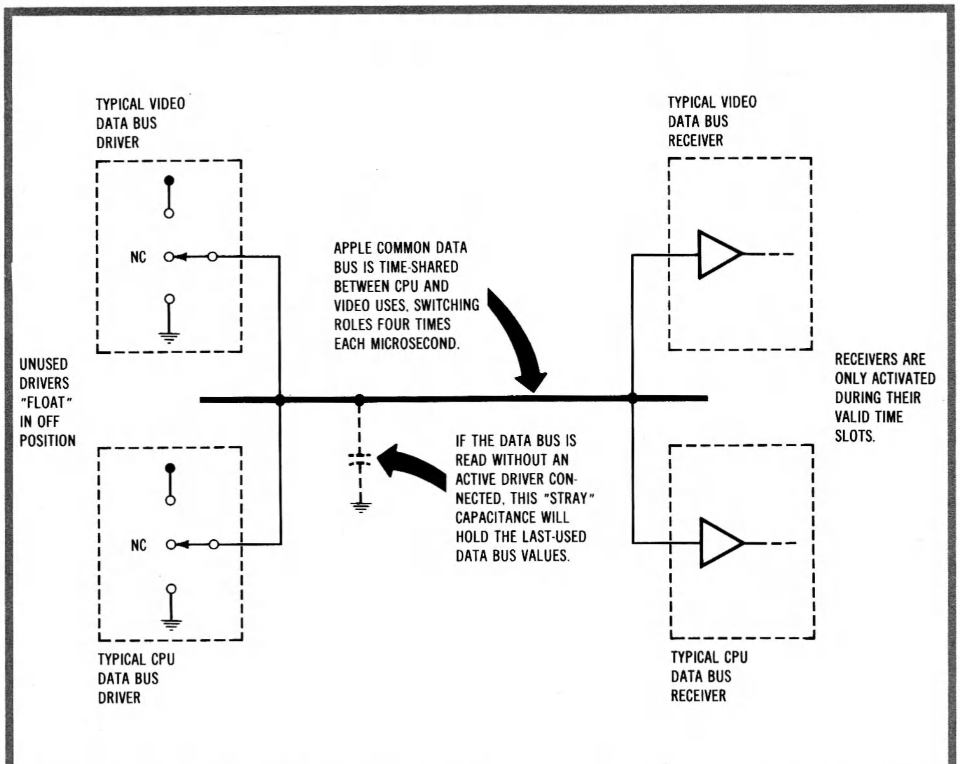


FIGURE 13-1. **Reading a "floating" Apple data bus will return the last video byte as valid data.**

Remember that the *same* internal Apple data bus is used *both* for video and CPU uses. The two take turns. Every one-half microsecond or so, an electronic switch, or multiplexer, gets flipped. The data bus is used for video for one-half microsecond, and then gets used for computing the next one-half microsecond. Video and CPU latches on the output grab the data bus during their intended time slots.

Thus, the video uses the data bus half the time, and the CPU uses the data bus the other half of the time. Careful timing keeps all of the video on the video side and all of the computing separately done on the CPU side.

Well, almost separate.

The data bus is activated by bus drivers. We've shown two typical bus drivers in Fig. 13-1. One bus driver is intended for video use, and a second one is intended for CPU use. These are called *tri-state* bus drivers. Tri-state bus drivers can connect a positive voltage to the bus for a logical ONE, can connect the bus to system ground for a logical ZERO, or can do nothing and just float there.

Each bus driver is shown as a three position, center off switch. You leave the bus driver in the "center," "unconnected," or "off" state when you do *not* use it. When you decide to use a bus driver, you briefly switch to the ONE or the ZERO state when your timing is just right. After forcing your data onto the data bus, you quickly switch back to the middle position.

Obviously, only one bus driver is allowed to be active at any time. All the others must float in their tri-state "off" positions.

The video bus drivers are activated as needed by the display timing. The CPU bus drivers are instead activated by the CPU. During a CPU memory *read* the bus driver is built into the RAM memory circuitry. Data will get sent *from* memory *to* CPU. During a memory *write* the bus driver will be built into the CPU. This time, data will get sent *from* CPU *to* memory.

Now for the neat part.

Most Apple memory locations are both written to and read from as a normal part of computer operation. But, certain Apple locations in the I/O space are set up to *only be written to,* while others are set up to *only be read from.*

If you try reading an "empty" Apple location, or if you try reading from an Apple location with "write-only" hardware in it, *nothing* will get connected to the data bus and you will end up with a floating data bus . . .

> Floating Data Bus
>
> A condition when a data bus is being read, but nothing is able to write to it.
>
> Software reading of a "write-only" memory location will attempt this.

Any electronic system has unavoidable stray capacitance in it. Since the Apple data bus is routed all over everywhere and since it connects to lots of different things, we can expect some fairly substantial stray data bus capacitance. In fact, there would be no way to completely avoid strays even if we tried.

Now, stray capacitance behaves just like any other capacitor. You have to charge the stray capacitance positive if you try to make the bus positive. And you have to discharge the stray capacitance if you try to lower the bus to ground. The bus drivers have to be powerful enough to quickly charge the circuit strays. If the drivers are not strong enough, delay-caused errors will result.

Normally, the next bus driver in use will quickly charge or discharge the data bus stray capacitance. But what happens when you *float* the data bus?

If you try this, the stray capacitance will act just like a circuit called a *sample and hold* that will *save* the last value put on the data bus. This happens because there is no new bus driver connected to quickly charge or discharge the stray capacitance.

So, if you read a floating data bus, you should be able to read the previous data value being temporarily saved for you by the stray bus capacitance.

The immediately previous use of the data bus was by the video. So any floating data bus read by the CPU will *pass on the next byte to be output as video* at that particular

instant. This happens because the stray capacitance "samples" and then "holds" the video value long enough for the CPU to accept the "held" value as valid data.

Try it and it works!

So . . .

> An Apple read of a floating data bus will return a clone of the next byte to be output as video.
>
> Once this clone is inside the CPU, it can be processed just like any other data value.

One good "write-only" location that produces a floating data bus is $C020, which is the cassette output location. The cassette output circuitry completely ignores the data bus. It simply recognizes its own private *address* on the *address* bus and toggles a flip-flop circuit whenever location $C020 is addressed. During a read of $C020, nothing is connected to the data bus. Since the data bus stray capacitance holds onto the last thing it received, the video byte from the previous one-half microsecond gets "caught" and held.

Magically, a byte intended to go out to the screen gets copied into the CPU for use or analysis . . .

> Commands such as . . .
>
> ```
> LDA $C020
> CMP$C020
> BIT  $C020
> ```
>
> (etc.)
>
> . . . will read the floating data bus and return a clone of the next video byte to be output.

The Vaporlock we are about to look at reads a floating data bus to move precise video timing information into the CPU.

This sounds sort of flakey, but the technique works very reliably on an Apple IIc or IIe with their fully buffered data buses. It works equally well on an Apple II+, so long as you don't plug a ridiculous number of cards into the expansion slots at once, or so long as you don't use certain oddball "problem" cards. More on this later.

Actually, the stray capacitance isn't really that small, and the amount of hold time isn't all that long. So, the Vaporlock would seem to be a reliable process that you can safely use in commercial programs. Best of all, the hardware-free Vaporlock lets us do a fast and exact lock that is completely free of jitter.

Being able to read what is going to the video screen is only half of the problem. Besides this, we have to somehow put just the right values into . . .

MAGIC SCREEN LOCATIONS

To use the Vaporlock, we have to put some magic bytes into some magic screen locations. Then we read and analyze the contents of these locations as they fly by on the way to the screen. A Vaporlock read to the floating data bus does this for us.

These magic screen locations must be output only during the blanking time. This way, we won't ever see them. The locations also have to be unique so they will only appear at one place on the screen. The magic locations also must be as near to the bottom of the screen as possible, if we are to do the fastest possible lock.

Finally, whatever we put into these magic locations must not appear elsewhere in the display in exactly the same order. If part of the display also maps the magic locations, you might get a false lock that will mess up the works. Regardless of the screen content, you must somehow prevent any false locks from happening.

Sounds tough.

And it is tough. But just possible. Fig. 13-2 shows us a block of magic memory locations near the bottom of the screen that have the needed magic properties.

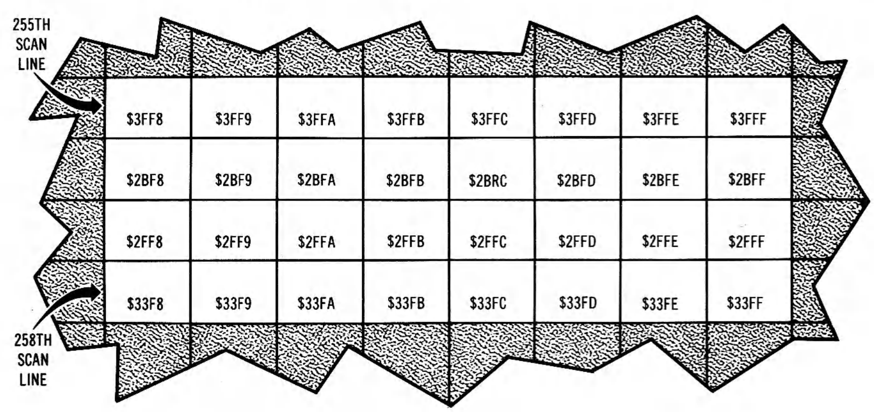


FIGURE 13-2. **This unique "i.d. patch" appears only once on the Apple screen during vertical blanking and retrace. Here are the magic addresses.**

We will always use the full HIRES page one when we begin our Vaporlock. Since what you see is what you get, your Apple must always pre-switch into this mode . . .

> For the Vaporlock to work —
>
> You MUST always be in the GRAPHICS, HIRES, PAGE ONE, and FULL-SCREEN modes at the instant you ask for an exact lock.

The HIRES full-page-one mode maps memory locations from $2000 though $3FFF onto the screen. Study the detailed maps in Enhancement 7 if you want to know what goes where. Mapping takes place continuously, even during the blank portions of the

screen. Continuous video addressing solves a refresh problem on the dynamic RAM memories, besides being simpler.

Many of the RAM locations are mapped twice, once on the "live" portion of the screen, and once on the blank portion. A few RAM locations are mapped *only* during the blank portions of the screen and thus are invisible and unused under normal circumstances. It turns out that memory locations ending in $78 through $7F or locations ending in $F8 through $FF are always invisible. Thus, $217A and $3AFC both get read from to the full HIRES1 screen, but are never seen.

Add them all up, and you'll find a total of 512 "invisible" memory locations. There's 16 per page for 32 pages from $2000 through $3FFF. These invisible memory locations are normally dangerous to use since they get plowed on every HIRES scrolling or screen clear.

What we need to find is a unique pattern of invisible memory locations that occurs only once on the screen. This also should happen as near the bottom of the screen as possible. We will call this unique pattern of invisible memory locations an i.d. patch . . .

> I.D. Patch —
>
> A unique group of invisible screen bytes that occur only once during a full HIRES screen scan.

So far, I have found only one totally unique i.d. patch. There is a rather violent change in system timing that takes place on the 255th consecutive scan in any field. Funny things happen between scan line 255 and scan line 256. These numbers assume the first scan at the top is called scan line 0. The funny things that happen only here let us set up the start of a unique i.d. patch.

Specifically, you will find the invisible addresses of $3FF8 through $3FFF on line 255 and, immediately below them one scan line later, you will find the invisible addresses of $2BF8 through $2BFF. Immediately below these, you will find addresses $2FF8 through $2FFF, and below those, you'll find $33F8 through $33FF.

Fig. 13-2 shows us the hidden addresses of the "magic" i.d. patch.

This i.d. patch is unique and occurs only in this one position, some six scan lines before the start of a new field. All we have to do now is decide what to put into the i.d. patch and how to use it.

Fig. 13-3 shows us one possible set of magic values that we can put in our i.d. patch. It turns out that we will only use 20 of the 32 possible i.d. bytes. To set up the Vaporlock, you pick six "magic" data values. You choose these data values as bytes that appear very rarely in typical HIRES pictures. The bytes are placed as shown.

Now, the trick is to (1) find the i.d. patch, (2) read the i.d. patch in such a way that jitter gets eliminated, and (3) be sure that nothing else on screen looks like the i.d. patch. A suitable software algorithm is shown in Fig. 13-4.

Finding the i.d. patch is easy enough. You simply search for the first i.d. value. In this example, you look for a value of $DB. If you go through the 6502's timing in detail, you'll find that there is an "inherent" seven clock cycle jitter in making a single search for one byte. This jitter happens because an LDA or CMP absolute takes four CPU cycles, and the test-and-try-again BNE takes another three. We can only make one test each seven CPU cycles as a minimum on a stock 6502. That is why there are seven DBs in the first row of the ID byte.

We catch one of the DBs in the top line of the i.d. patch. Which one? We don't know yet. But, having caught one of them, we delay exactly one horizontal line and

$DB $3FF8	$DB $3FF9	$DB $3FFA	$DB $3FFB	$DB $3FFC	$DB $3FFD	$DB $3FFE	— $3FFF
$24 $2BF8	$24 $2BF9	$24 $2BFA	$24 $2BFB	$B6 $2BRC	$B6 $2BFD	$B6 $2BFE	— $2BFF
$49 $2FF8	$49 $2FF9	$88 $2FFA	$88 $2FFB	— $2FFC	— $2FFD	— $2FFE	— $2FFF
$F3 $33F8	$F2 $33F9	— $33FA	— $33FB	— $33FC	— $33FD	— $33FE	— $33FF

FIGURE 13-3. **One possible set of magic data values for the i.d. patch.**

grab the next line of the i.d. patch. There are 65 clock cycles or roughly 65 microseconds in one Apple horizontal line. After a one-line delay, you grab a new value off the second line in the i.d. patch and analyze it.

Now the fun begins.

If we did *not* get a "hit" on an i.d. value of $24 or $B6, then we must have caught some *other* $DB somewhere else in the main display. This is unlikely, but if it happens, we go back and try again.

Remember that we could have hit any one of the seven $DBs on the first line of the i.d. patch. Let's call the first four of these *early* hits and the last three *late* hits. On an early hit, your second line i.d. value will be a $24. On a late hit, you'll read a $B6 instead.

You then delay *only* the early hits by four clock cycles.

Why?

To get rid of some of the jitter. Before, we could have ended up in any of seven character slots. Now we can only end up in four possible slots. We still have some jitter, but only half as much as we had before.

Go through the math, and you'll find one way to eliminate seven cycles of jitter is with three separate measurements each of which has two possible results. The trick is to get seven paths through the i.d. patch. Each path is arranged to take exactly the correct amount of time so that each path exits into the very same screen time slot.

Next, we delay for another line. More precisely, we delay for another line minus four cycles. We do this to move everything back into the "early" four slots on the third line of the i.d. patch. After this, we check those i.d. bytes again. If we do not get a $49 or a $88, then we must not be in the third line of the i.d. patch and have to go fetch some more.

This time, we can call catching one of the *first* two bytes early hits, and catching either of the *last* two bytes late hits. The early hits will be $49s, while the late ones will be $88s. Now, you delay the early hits by *two* cycles, removing half the remaining jitter. At this point, we have only one of two possible jitter values remaining.

Repeat the process one more time on the fourth and final line of the i.d. patch. If you get a $F3, stall one cycle. If, instead, you get a $F2 i.d. byte, do not stall. If you have neither a $F2 or a $F3, then go fetch again, because you must have somehow missed the patch.

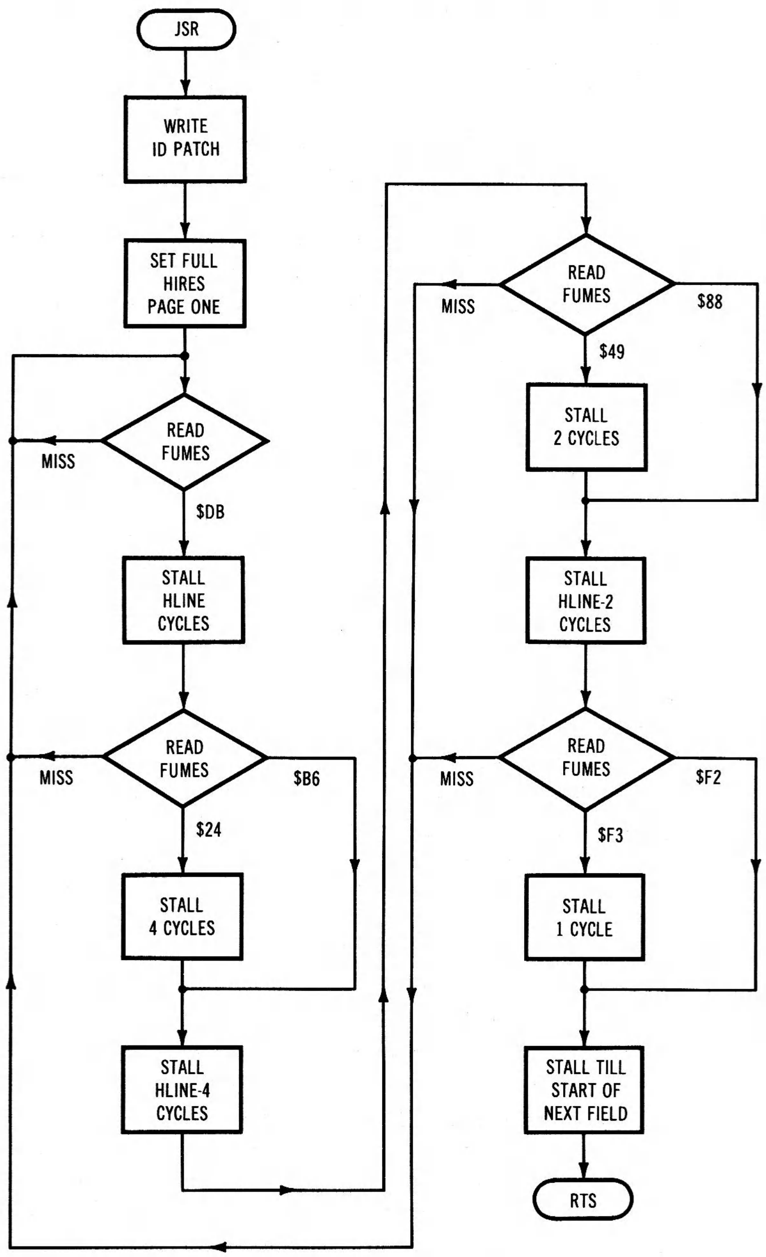


FIGURE 13-4. Flowchart shows algorithm used for exact Vaporlock synchronization.

At this point, if you caught each and every i.d. byte properly, you will have a jitter-free and exact screen lock. This locking occurs some three lines before the start of the next field. You can then burn up the remaining time with any old delay and phasing code of your choosing.

Reviewing, the first line of the i.d. patch lets you find the patch. The second line removes half the possible jitter. The third line removes another half of the remaining jitter. And the fourth line knocks out all the possible jitter that's left. For seven possible

hits on the first line, there are seven possible paths through the code. Each path differs by one microsecond. The "early" hits are delayed *more* than the late ones. Everything comes out even, and you end up with an exact lock and zero jitter.

One question remains. What are the odds of random HIRES bytes looking exactly like an i.d. patch and giving us a false lock? Well, we first have picked HIRES bytes that are seldom used. If some of them do not occur in your HIRES picture, then you are home free. Otherwise, you could change your i.d. bytes to be sure the patch is unique.

But, checking your HIRES pictures for certain bytes is not at all needed.

Say you have totally random HIRES data. This means you are pretty sure of a hit on the first i.d. byte. But you have to have one of two possible i.d. bytes exactly one line below, and then one of two new unlikely i.d. bytes below that, and finally one of yet two more unique i.d. bytes below that. The odds on a false lock? About (1/128) ↑ 3 or something like one screen in 2,097,152.

Taint likely, McGee.

So, the i.d. patch is essentially unique. You will never get a false lock on a HIRES picture unless you are *very* unlucky. Even with really bad luck, you can always change one i.d. byte and you are home free. Don't sweat it.

This particular i.d. patch scheme seems rather complicated. But remember that you have to both get an exact lock *and* reject false locks from random screen data. Let me know if you find a simpler i.d. patch and locking scheme that works as well as this one. Use either the response card or the hotline.

SOME CODE

Program 13-1 is a listing of the VAPORLOCK code, shown under EDASM. To handle the lowercase on "old" EDASM, just use *Apple Writer* IIe instead of the EDASM editor. The recent EDASM overhaul and upgrade now has lowercase built in and ready to go.

This EDASM program provides the code needed to find the start of a new video field, precisely and free from jitter.

We have put the code at $8AFF for those of you using HRCG on the DOS toolkit, but this code is relocatable more or less anywhere near an even page boundary *without* any need of reassembly.

Many more details on working with assembly language appear in my *Assembly Cookbook for the Apple II/IIe* (Sams Cat. No. 22331).

The Vaporlock appears on the companion diskette, both in EDASM assembler source code and as ready-to-use object code. To do an exact lock, you simply JSR to your Vaporlock. The Vaporlock then exits precisely and exactly on the start of the next field. The exit is jitter free. An exact lock takes as few as eight scan lines.

There are three parts to this code, called SETUP, LOCK, and STALL. In the SETUP portion, you write the magic i.d. values to the i.d. patch and then switch to HIRES full page one. This page switching is delayed on the IIe until the vertical blanking time. A check of an identification byte in the monitor verifies the version byte in the monitor at $FBE3. The delay eliminates a possible screen glitch on the IIe.

LOCK follows the algorithm of Fig. 13-4. The first line of the i.d. patch is sought out and found. The hit ends up with seven possible jitter values. The second line is then tested for validity and half the total jitter is removed. The third line is found, verified, and then is further de-jittered. Finally, the fourth line is found, verified, and the last possible jitter cycle is adjusted.

An exact and jitter-free lock is reached some three lines before the end of the present field. Should you miss one of the magic i.d. values, the Vaporlock assumes it is looking at the HIRES picture instead and starts over again.

PROGRAM 13-1 Listing of VAPORLOCK.SOURCE.

```
----- NEXT OBJECT FILE NAME IS VAPORLOCK

8AFF:        8AFF            ORG   $8AFF    ; FOR HIGH HRCG CHARACTER SET


8AFF:                  ;   *****************************************
8AFF:                  ;   *                                       *
8AFF:                  ;   *            -< VAPORLOCK >-            *
8AFF:                  ;   *                                       *
8AFF:                  ;   *      (FAST AND EXACT FIELD SYNC)      *
8AFF:                  ;   *                                       *
8AFF:                  ;   *       VERSION 1.0 ($8AFF-$8BC7)       *
8AFF:                  ;   *    for the APPLE II+ and APPLE IIe    *
8AFF:                  ;   *                                       *
8AFF:                  ;   *               10-12-83                *
8AFF:                  ;   *.......................................*
8AFF:                  ;   *                                       *
8AFF:                  ;   *          COPYRIGHT C 1983 BY          *
8AFF:                  ;   *                                       *
8AFF:                  ;   *     DON LANCASTER AND SYNERGETICS     *
8AFF:                  ;   *     BOX 1300, THATCHER AZ., 85552     *
8AFF:                  ;   *            (602) 428-4073             *
8AFF:                  ;   *                                       *
8AFF:                  ;   *     ALL COMMERCIAL RIGHTS RESERVED    *
8AFF:                  ;   *                                       *
8AFF:                  ;   *****************************************


8AFF:                  ;             *** WHAT IT DOES ***

8AFF:                  ;   This subroutine gives you a fast and exact field
8AFF:                  ;   sync that locks to the video screen in as few as
8AFF:                  ;   nine scan lines.  No hardware mods are needed.
8AFF:                  ;
8AFF:                  ;   The same code works on the Apple II+ or IIe.
8AFF:                  ;
8AFF:                  ;   The vaporlock, with suitable support software,
8AFF:                  ;   lets you mix and match HIRES, LORES, and text
8AFF:                  ;   anywhere on the screen, provides for glitchless
8AFF:                  ;   animation, simplifies light pens, allows grey
8AFF:                  ;   scale, text-over-color, professional video wipes,
8AFF:                  ;   and offers many other new visual display tricks
8AFF:                  ;   that seem "impossible" to do on a stock Apple.
8AFF:                  ;
8AFF:                  ;   Typical displays are totally free from any
8AFF:                  ;   glitches or jitter.


8AFF:                  ;             *** HOW TO USE IT ***

8AFF:                  ;   To lock to the video timing, do a JSR VAPORLK at
8AFF:                  ;   $8B00 or CALL 35840.
8AFF:                  ;
8AFF:                  ;   The vaporlock exits exactly and precisely on the
8AFF:                  ;   start of a new video field.
```

PROGRAM 13-1 cont

```
8AFF:             ;                *** GOTCHAS ***

8AFF:             ;   This code only runs on "real" Apples.
8AFF:             ;
8AFF:             ;   Franklins, clones, and look-alikes may have
8AFF:             ;   different timing that requires special code.
8AFF:             ;
8AFF:             ;   Certain oddball plug-in cards might interfere
8AFF:             ;   with operation on the II+.  Such interference
8AFF:             ;   is unlikley on the IIe.
8AFF:             ;
8AFF:             ;   Parts of the code have very critical timing
8AFF:             ;   and must not cross a page boundary.  If you
8AFF:             ;   relocate the code, put it all on one page.
8AFF:             ;

8AFF:             ;              *** ENHANCEMENTS ***

8AFF:             ; You can make a "phasing" adjustment by adding
8AFF:             ; or removing NOPs and branches in the PHASE code.
8AFF:             ; code.  Note that a NOP or a branch not taken uses
8AFF:             ; two clock cycles, while a branch taken needs three.
8AFF:             ;
8AFF:             ; You can preset the soft switches at the top of
8AFF:             ; the screen with suitable pokes to SHOW.
8AFF:             ;
8AFF:             ; VAPORLK object code is relocatable, if you put
8AFF:             ; it all on one page of memory.  Be sure to protect
8AFF:             ; memory and link to your first or second address.
8AFF:             ;

8AFF:             ;              *** RANDOM COMMENTS ***

8AFF:             ; The accumulator and all flags are saved to the
8AFF:             ; stack.  No use is made of the X or Y registers.
8AFF:             ;
8AFF:             ; The vaporlock exact field sync may be used in
8AFF:             ; your commercial programs provided fair credit
8AFF:             ; is prominently given.
8AFF:             ;
8AFF:             ; VAPORLK may be loaded as the highest HRCG
8AFF:             ; character set.
8AFF:             ;
8AFF:             ; Program length is $C8 (200) bytes.

8AFF:             ;            *** HOOKS ***

8AFF:       C052  FULL     EQU    $C052    ; FULL SCREEN SOFT SWITCH
8AFF:       C050  GR       EQU    $C050    ; GRAPHICS SOFT SWITCH
8AFF:       C057  HIRES    EQU    $C057    ; HIRES SOFT SWITCH
8AFF:       0006  IDBYTE   EQU    $06      ; ID VALUE FOR APPLE IIe
8AFF:       C054  PAGE1    EQU    $C054    ; PAGE ONE SOFT SWITCH
8AFF:       C020  SNIFF    EQU    $C020    ; FLOATING DATA BUS READ ADDRESS
8AFF:       C051  TEXT     EQU    $C051    ; TEXT SOFT SWITCH
8AFF:       C019  VBLANK   EQU    $C019    ; JITTERY V BLANKING (IIe ONLY!)
8AFF:       FBB3  VERSION  EQU    $FBB3    ; SYSTEM ID BYTE LOCATION
8AFF:       FCA8  WAIT     EQU    $FCA8    ; MONITOR DELAY SUBROUTINE
```

PROGRAM 13-1 cont

```
8AFF:              114 ;            *** CONSTANTS ***

8AFF:        00DB  116 ID0      EQU   $DB        ; ID BYTES FOR SYNC PATCH
8AFF:        0024  117 ID1      EQU   $24        ;  (ALL SHOULD BE RARELY USED)
8AFF:        00B6  118 ID2      EQU   $B6        ;
8AFF:        0049  119 ID3      EQU   $49        ;
8AFF:        0088  120 ID4      EQU   $88        ;
8AFF:        00F3  121 ID5      EQU   $F3        ; THIS BYTE MUST BE ODD VALUE!

8AFF:              124 ;          *** VAPORLOCK SUBROUTINE ****

8AFF:              126 ;  There are three parts to the Vaporlock
8AFF:              127 ;  subroutine.  These are SETUP, LOCK, and STALL.
8AFF:              128 ;
8AFF:              129 ;  SETUP works by writing a magic "id patch"
8AFF:              130 ;  to invisible locations on the text screen.
8AFF:              131 ;  These magic locations tap the unique 3FFX
8AFF:              132 ;  to 2BFX transitions that happen only on
8AFF:              133 ;  the invisible advance from line 255 to 256.
8AFF:              134 ;
8AFF:              135 ;  SETUP also forces the full HIRES1 mode
8AFF:              136 ;  during the locking process.  A IIe-only
8AFF:              137 ;  blanking search minimizes any entry glitches.
8AFF:              138 ;
8AFF:              139 ;  LOCK searches for the magic combinations
8AFF:              140 ;  of invisible ID bytes, starting on line
8AFF:              141 ;  255.  Four lines are needed for complete
8AFF:              142 ;  and exact locking.  One half of the
8AFF:              143 ;  possible jitter is eliminated on each of
8AFF:              144 ;  the second, third, and fourth lines,
8AFF:              145 ;  ending with an exact lock at the end of
8AFF:              146 ;  blank screen line 258.
8AFF:              147 ;
8AFF:              148 ;  LOCK uses the "floating data bus" read
8AFF:              149 ;  technique pioneered by Bob Bishop.  If
8AFF:              150 ;  an Apple location is read addressed in
8AFF:              151 ;  which there is no read hardware, a
8AFF:              152 ;  floating data bus results.  This floating
8AFF:              153 ;  data bus acts as a "sample and hold" that
8AFF:              154 ;  saves the last video screen access.  The
8AFF:              155 ;  "fumes" that remain from the previous
8AFF:              156 ;  video screen access can be read as data.
8AFF:              157 ;
8AFF:              158 ;  STALL delays until the exact start of
8AFF:              159 ;  the field.  It is presently set up
8AFF:              160 ;  to exit exactly on the start of the live
8AFF:              161 ;  screen at the top of the field.  You
8AFF:              162 ;  can adjust this for phasing, or to gain
8AFF:              163 ;  pre-screen time for setup or actions.
8AFF:              164 ;  The exit is exact and jitter-free.
8AFF:              165 ;
8AFF:              166 ;  The FIX2+ routine provides one extra
8AFF:              167 ;  delay cycle to adjust for screen switching
8AFF:              168 ;  differences between the IIe and II+.
8AFF:              169 ;  For II+ exotic (non-screen) field switching,
8AFF:              170 ;  you might want to defeat this adjustment.
```

PROGRAM 13-1 cont

```
8AFF:                ;            ** SETUP **

8AFF:EA                      NOP              ; EQUALIZE TO PAGE BOUNDARY
8B00:08                VAPORLK PHP            ; SAVE FLAGS
8B01:48                      PHA              ; SAVE ACCUMULATOR

8B02:A9 DB                   LDA    #ID0      ; WRITE ID PATCH
8B04:8D F8 3F                STA    $3FF8     ;  TO LINE #255
8B07:8D F9 3F                STA    $3FF9     ;
8B0A:8D FA 3F                STA    $3FFA     ;
8B0D:8D FB 3F                STA    $3FFB     ;
8B10:8D FC 3F                STA    $3FFC     ;
8B13:8D FD 3F                STA    $3FFD     ;
8B16:8D FE 3F                STA    $3FFE     ;

8B19:A9 24                   LDA    #ID1      ;  TO LINE #256
8B1B:8D F8 2B                STA    $2BF8     ;
8B1E:8D F9 2B                STA    $2BF9     ;
8B21:8D FA 2B                STA    $2BFA     ;
8B24:8D FB 2B                STA    $2BFB     ;
8B27:A9 B6                   LDA    #ID2      ;
8B29:8D FC 2B                STA    $2BFC     ;
8B2C:8D FD 2B                STA    $2BFD     ;
8B2F:8D FE 2B                STA    $2BFE     ;

8B32:A9 49                   LDA    #ID3      ;  TO LINE 257
8B34:8D F8 2F                STA    $2FF8     ;
8B37:8D F9 2F                STA    $2FF9     ;
8B3A:A9 88                   LDA    #ID4      ;
8B3C:8D FA 2F                STA    $2FFA     ;
8B3F:8D FB 2F                STA    $2FFB     ;

8B42:A9 F3                   LDA    #ID5      ; AND FINALLY TO LINE 258
8B44:8D F8 33                STA    $33F8     ;
8B47:A9 F2                   LDA    #ID5-1    ;
8B49:8D F9 33                STA    $33F9     ;

8B4C:A9 06                   LDA    #IDBYTE   ; CHECK FOR A IIe
8B4E:CD B3 FB                CMP    VERSION   ;
8B51:D0 05      8B58         BNE    MORE0     ;
8B53:2C 19 C0          VBFIND BIT   VBLANK    ; WAIT TILL IIe BLANKING START
8B56:30 FB      8B53         BMI    VBFIND    ;

8B58:2C 50 C0          MORE0 BIT    GR        ; FORCE FULL HIRES PAGE ONE
8B5B:2C 57 C0                BIT    HIRES     ;
8B5E:2C 52 C0                BIT    FULL      ;
8B61:2C 54 C0                BIT    PAGE1     ; THEN FALL THROUGH TO LOCK


8B64:                ;            ** LOCK **


8B64:A9 DB             LOCK  LDA    #ID0      ; LOOK FOR FIRST PATCH ID VALUE
8B66:CD 20 C0          RETRY1 CMP   SNIFF     ;
8B69:D0 FB      8B66         BNE    RETRY1    ;
8B6B:A9 02                   LDA    #02       ; DELAY FOR EXACTLY 57 CYCLES
8B6D:20 A8 FC                JSR    WAIT      ; (HLINE-BNE-LDA#-LDA)
8B70:48                      PHA              ;
```

PROGRAM 13-1 cont

```
8B71:68                  PLA              ;
8B72:AD 20 C0            LDA   SNIFF      ; GET SECOND PATCH ID VALUE
8B75:C9 B6               CMP   #ID2       ; JITTER 4,5,OR 6?
8B77:F0 06     8B7F      BEQ   MORE1      ;
8B79:C9 24               CMP   #ID1       ; JITTER 0,1,2, OR 3?
8B7B:F0 02     8B7F      BEQ   MORE1      ; OK TO CONTINUE
8B7D:D0 E5     8B64      BNE   LOCK       ; MISSED, TRY AGAIN

8B7F:A9 02          MORE1 LDA  #02        ; DELAY FOR EXACTLY 50 CYCLES
8B81:20 A8 FC            JSR   WAIT       ;  (HLINE-4-CMP#-BEQ-LDA#-LDA)
8B84:AD 20 C0            LDA   SNIFF      ; GET THIRD PATCH ID VALUE
8B87:C9 88               CMP   #ID4       ; JITTER 2 OR 3?
8B89:F0 04     8B8F      BEQ   MORE2      ; YES
8B8B:C9 49               CMP   #ID3       ; JITTER 0 OR 1
8B8D:F0 02     8B91      BEQ   MORE3      ; ONLY WANT 2 CLOCK CORRECTION

8B8F:D0 D3     8B64 MORE2 BNE  LOCK       ; CURSES! FOILED AGAIN!
8B91:A9 02          MORE3 LDA  #$02       ; DELAY FOR EXACTLY 50 CYCLES
8B93:20 A8 FC            JSR   WAIT       ;  (HLINE-2-CMP#-BEQ-BNE-LDA#-LDA)
8B96:AD 20 C0            LDA   SNIFF      ; GET FOURTH PATCH ID VALUE
8B99:4A                  LSR   A          ; SHIFT INTO CARRY
8B9A:B0 00     8B9C      BCS   MORE4      ; TO EQUALIZE ONE COUNT

8B9C:C9 79          MORE4 CMP  #ID5/2     ; FINAL VALIDITY CHECK
8B9E:D0 C4     8B64      BNE   LOCK       ; BACK TO SQUARE ONE
8BA0:EA                  NOP              ; HAVE LOCK AT THIS POINT



8BA1:               ;              ** STALL **


8BA1:A9 05          STALL LDA  #05        ; DELAY FOR EXACTLY 193 CYCLES
8BA3:20 A8 FC            JSR   WAIT       ;
8BA6:A9 02               LDA   #02        ;
8BA8:20 A8 FC            JSR   WAIT       ;

8BAB:A9 06          FIX2+ LDA  #IDBYTE    ; ADD ONE EXTRA CYCLE ONLY ON
8BAD:CD B3 FB            CMP   VERSION    ;  THE II+ TO EQUALIZE ON-SCREEN
8BB0:D0 00     8BB2      BNE   SHOW       ;  DISPLAY MODE SWITCHING

8BB2:2C 20 C0       SHOW  BIT  SNIFF      ; OPTIONAL MODE CHANGES GO HERE
8BB5:2C 20 C0            BIT   SNIFF      ;
8BB8:2C 20 C0            BIT   SNIFF      ;
8BBB:2C 20 C0            BIT   SNIFF      ;

8BBE:18             PHASE CLC             ; PHASING CHANGES GO HERE
8BBF:B0 00     8BC1      BCS   MORE5      ;
8BC1:EA             MORE5 NOP             ; EACH BRANCH TAKEN = 3
8BC2:EA                  NOP              ; EACH BRANCH NOT TAKEN = 2
8BC3:EA                  NOP              ; EACH NOP = 2 CHARACTERS
8BC4:EA                  NOP              ;

8BC5:68                  PLA              ; RESTORE ACCUMULATOR AND FLAGS
8BC6:28                  PLP              ;
8BC7:60                  RTS              ; AND EXIT
```

STALL does just that. It burns up enough cycles to exit you precisely on the start of the new field. The first part of STALL uses two trips through the monitor WAIT subroutine at $FCA8. By the way, a bonus program called the Triple Delay Finder is provided on the companion diskette. This program can be used to quickly find most any single, double, or triple trip through the WAIT code.

The Apple II+ switches its screen modes one cycle earlier than the IIe. A routine called FIX2+ is included that makes the screen exit appear the same on both versions of the Apple. A service module called SHOW follows next. SHOW can be used to preset what will get displayed as the next scan begins. For instance, a $2C $56 $C0 puts you into LORES, and so on down the usual list.

A final routine is called PHASE. This lets you advance or retard the exit of Vaporlock before or after the exact field start by as many cycles as you want. This is handy for doing field switching at oddball locations on the screen. Video wipes usually will need this phasing adjustment.

Another use of PHASE lets you call the Vaporlock as a subroutine, rather than building it into your code. Just exit the Vaporlock *six* cycles short and add an RTS to the end of the module. You will now return to your main code with an exact lock at the exact screen start. This is handy if you have several routines that separately will need an exact lock.

Three test routines are shown you in Chart 13-1. The first is a quick Vaporlock tester called VLTEST. This gives you a split-screen display with text on the top half and HIRES on the bottom. Most importantly, a single HIRES line four-characters long appears at extreme upper right. You can use this for locking and phasing experiments.

The second test routine is named ALTERNATER. This field alternater flips you between text on one complete field and HIRES on the next complete field. As usual, field alternaters will give you some flicker. Practically all other mixed-field uses are completely glitch and flicker free.

I almost didn't include a field alternater here since they can do so many bad things in such awful ways. You can minimize flicker by keeping as much of the screen as possible either black or constant, by using darker colors, setting minimum contrast, or, best of all, by using an orange long-persistence display monitor. The reason to show an alternater at all is to get you thinking about all of the new possibilities of exact field sync, however bad some of them might seem.

If you do use an amber screen, the flicker is nearly invisible. Unfortunately, all of those new amber screens are totally unsuited for most games or animated graphics. The same persistence that helps the alternater destroys animation by changing balls into comets, blurring motion, and doing other bad things.

UPDATING VFFS FILES

The Vaporlock by itself is only a service routine that exits you exactly on the start of a field. Very nicely, you can now relock on each and every field. This leaves time for you to do all sorts of other things during or after a mixed-field display. It also makes your software much simpler, since you no longer have to hold exact timing values for the entire display time.

For instance, you can now do a mixed-field display on one field and return to your main program at the end of the field mixing. You can then do other things before you grab the next lock. Timing is not at all critical, so long as you finish up whatever it is you want to do before the minimum locking time on the next field. For most mixed-field uses, you have several thousand clock cycles available for your use per field without serious restrictions.

Chart 13-1. VAPORLOCK Test Utilities

NOTE: For these tests to be useful, you should place something of interest both on text page one and on HIRES page one. Results will not be obvious with an all-black HIRES screen or a completely cleared text screen.

I. VAPORLOCK tester:

```
] BLOAD VAPORLOCK <cr>
] CALL -151 <cr>
* 2000: FF FF FF FF FF FF FF <cr>
* 0300: 20 00 8B 2C 51 C0 A9 2D <cr>
* 0308: 20 A8 FC 2C 50 C0 4C 00 03 <cr>
* 0300G <cr>
```

You should get a split-screen display, top half text, bottom half HIRES. A single HIRES line four characters long should show in the extreme upper left. The name of this utility is VLTEST.

II. Field Alternater:

```
] BLOAD VAPORLOCK <cr>
] CALL -151 <cr>
* 0320: A9 50 8D BC 8B 20 00 8B <cr>
* 0328: A9 51 8D BC 8B 20 00 8B 4C 20 03 <cr>
* 0320G <cr>
```

You should see superimposed text and HIRES pages. The wretched flickering can sometimes be gotten around by using an amber monitor or through careful choice of program material and display settings. See text for more details. The name of this utility is ALTERNATER.

III. VLFFS Window Tester:

```
] BLOAD VLFFS.EMPTY <cr>
] CALL -151 <cr>
* 8BA0: 50 <cr>
* 8BAF: 51 <cr>
* 8CA0: E0 <cr>
* 8C44: E0 <cr>
* 8CFB: 80 <cr>
* 8DC4: 51 <cr>
* 8B00G <cr>
```

You should see a text screen with an inset HIRES window. The name of this utility is VLFFS.WINDOW.

These and many other demos and use examples appear on the companion diskette to this volume.

In fact, for some field mixers, you can retain 90 percent or higher throughput. This can happen if you only mix fields on the first few lines of a display, such as a graph title or a scoring clock inset into a HIRES game display.

A program called VFFS.EMPTY was shown you in Enhancement 5 that let you mix and match HIRES, LORES, and text anywhere on the screen in any combination. A new program called VLFFS.EMPTY is shown as Program 13-2. This program does everything that VFFS.EMPTY does, but now includes the Vaporlock exact lock. The code for exiting on keypressed and timeout is still built in. You also have the new option of doing anything you like with the hook URCODE, so long as whatever you are doing only lasts a few thousand clock cycles.

This EDASM program will let you mix and match text, HIRES, and LORES anywhere on the screen. The exact display format is changed by altering the VPAT and HPAT files.

There are two different ways to use VLFFS.EMPTY. If you enter the code at ONESCAN, you get a single mixed field that lasts for one tv scan from top to bottom. You then return to your main program. If you enter the code at CONSCAN, the mixed fields keep going round and round until timeout or a pressed key. Even with CONSCAN, you can use several thousand clock cycles at URCODE anyway you wish.

Once again, URCODE timing is not at all critical. Just be sure to end it in time for the next lock on the next screen.

VLFFS.EMPTY has been carefully set up to keep the same entry and file points as VFFS.EMPTY. All of the work sheets and file details of Enhancement 5 may still be used for VLFFS.EMPTY. Be sure to refer back to this older Enhancement before you try using any mixed fields.

A most interesting tester for VLFFS.EMPTY is shown as the third test program in Chart 13-1. It's called VLFFS.WINDOW and will give you a small HIRES window completely surrounded by text. Other windows can easily be added wherever you like. Sizes are also easily changed.

You can update any existing VFFS.WHATEVER files you may have already created with Program 13-3, the VFFS>VLFFS CONVERTER. This program is self-prompting and automatic. A few hand patches to SHOW may be needed for some uses. For instance, VFFS.GIRLS needs LORES affirmed at the top of the screen and VFFS.BYE needs mixed graphics affirmed.

The previous demo of FUN WITH MIXED FIELDS has been updated to FUN WITH VLFFS FIELDS and appears on the support diskette for this volume. LORES COLORS 121 has also been upgraded to LORES COLORS 121 (VL) and also appears on the companion diskette.

Should you want to shorten the mixed-field part of your display for more computing throughput, just change NEWFLD. A value of $C0 gives you a full screen of mixed fields. Use $A0 to provide four lines of text at the bottom, $80 for eight text lines, and so on. A value of $08 just lets you mix the top eight scan lines. Anything between gives you whatever tradeoff you want, swapping available computing time against mixed-field displaying.

PROGRAM 13-2 Listing of VLFFS.EMPTY.SOURCE.

```
----- NEXT OBJECT FILE NAME IS VLFFS.EMPTY

8AFF:        8AFF         ORG  $8AFF     ; FOR HIGHEST HRCG CHARACTER SET


8AFF:                ;  *****************************************
8AFF:                ;  *                                       *
8AFF:                ;  *           -< VLFFS.EMPTY >-           *
8AFF:                ;  *                                       *
8AFF:                ;  *     (VAPORLOCK FIELD FORMATTER SUB)   *
8AFF:                ;  *                                       *
8AFF:                ;  *        VERSION 1.0 ($8AFF-$8DCE)      *
8AFF:                ;  *     for the APPLE II+ and APPLE IIe   *
8AFF:                ;  *                                       *
8AFF:                ;  *               10-12-83                *
8AFF:                ;  *.......................................*
8AFF:                ;  *                                       *
8AFF:                ;  *          COPYRIGHT C 1983 BY          *
8AFF:                ;  *                                       *
8AFF:                ;  *      DON LANCASTER AND SYNERGETICS    *
8AFF:                ;  *      BOX 1300, THATCHER AZ., 85552    *
8AFF:                ;  *           (602) 428-4073              *
8AFF:                ;  *                                       *
8AFF:                ;  *      ALL COMMERCIAL RIGHTS RESERVED   *
8AFF:                ;  *                                       *
8AFF:                ;  *****************************************


8AFF:                ;          *** WHAT IT DOES ***

8AFF:                ;  This subroutine lets you mix and match HIRES,
8AFF:                ;  LORES, and text anywhere on the screen.
8AFF:                ;
8AFF:                ;  The screen mode can be changed once each HBLANK
8AFF:                ;  time using the VPATRN file, and up to ten times
8AFF:                ;  per H scan using files HPAT1 through HPAT4.
8AFF:                ;
8AFF:                ;  VLFFS displays are totally glitch and flicker free.
8AFF:                ;
8AFF:                ;  See Enhancements #5 and #13 of ENHANCING YOUR
8AFF:                ;  APPLE II for full use details.


8AFF:                ;          *** HOW TO USE IT ***

8AFF:                ;  To use, set up your pattern files ahead of
8AFF:                ;  time, deciding which screen switches get
8AFF:                ;  flipped where on the screen.
8AFF:                ;
8AFF:                ;  For a single field scan,JSRONESCAN at
8AFF:                ;  $8B21 or CALL 35617.
8AFF:                ;
8AFF:                ;  For continuous display till timeout or
8AFF:                ;  a key is pressed,JSRCONSCAN at $8B00
8AFF:                ;  or CALL 35584.
```

PROGRAM 13-2 cont

```
8AFF:               ;              *** GOTCHAS ***

8AFF:               ;   This code only runs on "real" Apples.
8AFF:               ;
8AFF:               ;   Franklins, clones, and look-alikes may have
8AFF:               ;   different timing that requires special code.
8AFF:               ;
8AFF:               ;   Certain oddball plug-in cards might interfere
8AFF:               ;   with operation on the II+.  Such interference
8AFF:               ;   is unlikley on the IIe.
8AFF:               ;
8AFF:               ;   Parts of the code have very critical timing
8AFF:               ;   and must not cross a page boundary.  Watch
8AFF:               ;   this detail on any code relocation.
8AFF:               ;
8AFF:               ;   A maximum of four HPAT files are presently
8AFF:               ;   allowed.  These can repeat in sequence as
8AFF:               ;   often as needed per screen.


8AFF:               ;            *** ENHANCEMENTS ***

8AFF:               ;  You can make a "phasing" adjustment by adding
8AFF:               ;  or removing NOPs and branches at PHASE.  Note
8AFF:               ;  that a NOP or a branch not taken uses two clock
8AFF:               ;  cycles, while a branch taken needs three.
8AFF:               ;
8AFF:               ;  You can preset the soft switches at the top of
8AFF:               ;  the screen with suitable pokes to SHOW.
8AFF:               ;
8AFF:               ;  Several thousand machine cycles per field are
8AFF:               ;  available for your use per field.  These need not
8AFF:               ;  be synchronized, and timing is not critical.
8AFF:               ;
8AFF:               ;  The number of field switched lines per scan is
8AFF:               ;  set by NEWFLD+1 at $8B25 (35621).


8AFF:               ;            *** RANDOM COMMENTS ***

8AFF:               ;  The VLFFS.EMPTY exact field formatter may be used
8AFF:               ;  in your commercial programs provided fair credit
8AFF:               ;  is prominently given.
8AFF:               ;
8AFF:               ;  VFSS Worksheets 5-1, 5-2, and 5-3 may be used with
8AFF:               ;  VLFFS.  VFFS programs need conversion to VLFFS.
8AFF:               ;
8AFF:               ;  Be sure to carefully study Enhancements #5 and
8AFF:               ;  #13 before attempting to use this code.
8AFF:               ;
8AFF:               ;  Program length is $2CF (719) bytes.


8AFF:               ;            *** HOOKS ***

8AFF:        C060     DUMMY    EQU   $C060     ; LOCATION FOR NO-SWITCH H
8AFF:        C052     FULL     EQU   $C052     ; FULL SCREEN SOFT SWITCH
8AFF:        C050     GR       EQU   $C050     ; GRAPHICS SOFT SWITCH
```

PROGRAM 13-2 cont

```
8AFF:        C057     HIRES    EQU  $C057    ; HIRES SOFT SWITCH
8AFF:        0006     IDBYTE   EQU  $06      ; ID VALUE FOR APPLE IIe
8AFF:        C010     KBSTRB   EQU  $C010    ; KEYBOARD STROBE RESET
8AFF:        C000     KEYBD    EQU  $C000    ; KEY PRESS CHECK
8AFF:        C054     PAGE1    EQU  $C054    ; PAGE ONE SOFT SWITCH
8AFF:        C020     SNIFF    EQU  $C020    ; FLOATING DATA BUS READ ADDRESS
8AFF:        C000     SWITCH   EQU  $C000    ; VFILE SWITCH LOCATION
8AFF:        C051     TEXT     EQU  $C051    ; TEXT SOFT SWITCH
8AFF:        C019     VBLANK   EQU  $C019    ; JITTERY V BLANKING (IIe ONLY!)
8AFF:        FBB3     VERSION  EQU  $FBB3    ; SYSTEM ID BYTE LOCATION
8AFF:        FCA8     WAIT     EQU  $FCA8    ; MONITOR DELAY SUBROUTINE

8AFF:                 ;             *** CONSTANTS ***

8AFF:        00DB     ID0      EQU  $DB      ; ID BYTES FOR SYNC PATCH
8AFF:        0024     ID1      EQU  $24      ;  (ALL MUST BE RARE HIRES BYTES)
8AFF:        00B6     ID2      EQU  $B6      ;
8AFF:        0049     ID3      EQU  $49      ;
8AFF:        0088     ID4      EQU  $88      ;
8AFF:        00F3     ID5      EQU  $F3      ; THIS BYTE MUST BE AN ODD VALUE!

8AFF:                 ;           **** VLFS.EMPTY ****

8AFF:                 ;  There are four parts to the VLFFS.EMPTY subroutine.
8AFF:                 ;  These include CONSCAN, ONESCAN,  VAPORLK, and URCODE.
8AFF:                 ;
8AFF:                 ;  CONSCAN continuously exercises the field formatting
8AFF:                 ;  until timeout or a key is pressed.
8AFF:                 ;
8AFF:                 ;  ONESCAN is the old VFFS.EMPTY modified to work only
8AFF:                 ;  for a single field.
8AFF:                 ;
8AFF:                 ;  VAPORLK is the vaporlock exact field sync code,
8AFF:                 ;  shortened slightly for VFFS.EMPTY compatibility.
8AFF:                 ;
8AFF:                 ;  URCODE is a hook that lets you do anything you want
8AFF:                 ;  for up to several thousand clock cycles per field.
8AFF:                 ;

8AFF:                 ;           ** CONTINUOUS SCANNER **

8AFF:EA               ST       NOP           ; EVEN PAGE START (FOR HRCG)

8B00:20 FE 8C         CONSCAN  JSR  SETUP    ; SET TIMEOUT VALUE
8B03:20 21 8B         AGAIN1   JSR  ONESCAN  ; DO A SINGLE SCAN
8B06:20 0C 8B                  JSR  URCODE   ; DO WHATEVER YOU HAVE TIME FOR
8B09:90 F8     8B03            BCC  AGAIN1   ; REPEAT SCAN TILL TIMEOUT OR KP
8B0B:60                        RTS           ; EXIT ON SET CARRY

8B0C:60               URCODE   RTS           ; RETURN IF NO USER CODE
```

PROGRAM 13-2 cont

```
8B0D:                 163 ;           ** SINGLE FIELD SCAN **

8B21:          8B21 165          ORG  ST+$22     ; BURN BYTES TO MATCH VFFSY
8B21:20 08 8D         166 ONESCAN JSR  VAPORLK    ; DO EXACT LOCK TO EDGE


8B24:                 169 ;           ** START OF FIELD **


8B24:A0 C0            171 NEWFLD  LDY  #$C0       ; FOR 192 LINES

8B26:B9 00 8C         173 NXTLN1  LDA  VPATRN,Y  ; GET LINE PATTERN
8B29:30 63     8B8E 174          BMI  HPAT2      ;
8B2B:10 00     8B2D 175 HPAT1   BPL  HP1        ;
8B2D:29 7F            176 HP1     AND  #$7F       ; MASK SWITCH COMMAND
8B2F:AA               177         TAX             ;
8B30:9D 00 C0         178         STA  SWITCH,X  ;
8B33:8D 60 C0         179         STA  DUMMY      ;
8B36:8D 60 C0         180         STA  DUMMY      ;
8B39:8D 60 C0         181         STA  DUMMY      ;     *
8B3C:8D 60 C0         182         STA  DUMMY      ;    **
8B3F:8D 60 C0         183         STA  DUMMY      ;     *
8B42:8D 60 C0         184         STA  DUMMY      ;     *
8B45:8D 60 C0         185         STA  DUMMY      ;     *
8B48:8D 60 C0         186         STA  DUMMY      ;     *
8B4B:8D 60 C0         187         STA  DUMMY      ;    ***
8B4E:8D 60 C0         188         STA  DUMMY      ;
8B51:88               189         DEY             ; ONE LESS LINE
8B52:F0 32     8B86 190          BEQ  BOTTOM     ; AT SCREEN BOTTOM?
8B54:D0 D0     8B26 191          BNE  NXTLN1     ;

8B56:B9 00 8C         193 NXTLN4  LDA  VPATRN,Y  ; GET LINE PATTERN
8B59:30 D0     8B2B 194          BMI  HPAT1      ;
8B5B:10 00     8B5D 195 HPAT4   BPL  HP4        ;
8B5D:29 7F            196 HP4     AND  #$7F       ; MASK SWITCH COMMAND
8B5F:AA               197         TAX             ;
8B60:9D 00 C0         198         STA  SWITCH,X  ;
8B63:8D 60 C0         199         STA  DUMMY      ;
8B66:8D 60 C0         200         STA  DUMMY      ;
8B69:8D 60 C0         201         STA  DUMMY      ;      *
8B6C:8D 60 C0         202         STA  DUMMY      ;     **
8B6F:8D 60 C0         203         STA  DUMMY      ;    * *
8B72:8D 60 C0         204         STA  DUMMY      ;   *  *
8B75:8D 60 C0         205         STA  DUMMY      ;   *****
8B78:8D 60 C0         206         STA  DUMMY      ;      *
8B7B:8D 60 C0         207         STA  DUMMY      ;      *
8B7E:8D 60 C0         208         STA  DUMMY      ;
8B81:88               209         DEY             ; ONE LESS LINE
8B82:F0 02     8B86 210          BEQ  BOTTOM     ; AT SCREEN BOTTOM?
8B84:D0 D0     8B56 211          BNE  NXTLN4     ;

8B86:4C C2 8C         213 BOTTOM  JMP  BTM        ; "SPLICE" RELATIVE BRANCH

8B89:B9 00 8C         216 NXTLN2  LDA  VPATRN,Y  ; GET LINE PATTERN
8B8C:30 30     8BBE 217          BMI  HPAT3      ;
8B8E:10 00     8B90 218 HPAT2   BPL  HP2        ;
8B90:29 7F            219 HP2     AND  #$7F       ; MASK SWITCH COMMAND
8B92:AA               220         TAX             ;
8B93:9D 00 C0         221         STA  SWITCH,X  ;
```

PROGRAM 13-2 cont

```
8B96:8D 60 C0        222          STA  DUMMY    ;
8B99:8D 60 C0        223          STA  DUMMY    ;
8B9C:8D 60 C0        224          STA  DUMMY    ;     ***
8B9F:8D 60 C0        225          STA  DUMMY    ;    *   *
8BA2:8D 60 C0        226          STA  DUMMY    ;        *
8BA5:8D 60 C0        227          STA  DUMMY    ;       *
8BA8:8D 60 C0        228          STA  DUMMY    ;     *
8BAB:8D 60 C0        229          STA  DUMMY    ;    *
8BAE:8D 60 C0        230          STA  DUMMY    ;    ******
8BB1:8D 60 C0        231          STA  DUMMY    ;
8BB4:88              232          DEY           ; ONE LESS LINE
8BB5:F0 CF     8B86  233          BEQ  BOTTOM   ; AT SCREEN BOTTOM?
8BB7:D0 D0     8B89  234          BNE  NXTLN2   ;

8BB9:B9 00 8C        236 NXTLN3   LDA  VPATRN,Y ; GET LINE PATTERN
8BBC:30 9D     8B5B  237          BMI  HPAT4    ;
8BBE:10 00     8BC0  238 HPAT3    BPL  HP3      ;
8BC0:29 7F           239 HP3      AND  #$7F     ; MASK SWITCH COMMAND
8BC2:AA              240          TAX           ;
8BC3:9D 00 C0        241          STA  SWITCH,X ;
8BC6:8D 60 C0        242          STA  DUMMY    ;
8BC9:8D 60 C0        243          STA  DUMMY    ;
8BCC:8D 60 C0        244          STA  DUMMY    ;   *****
8BCF:8D 60 C0        245          STA  DUMMY    ;       *
8BD2:8D 60 C0        246          STA  DUMMY    ;      *
8BD5:8D 60 C0        247          STA  DUMMY    ;     **
8BD8:8D 60 C0        248          STA  DUMMY    ;       *
8BDB:8D 60 C0        249          STA  DUMMY    ;   *   *
8BDE:8D 60 C0        250          STA  DUMMY    ;    ***
8BE1:8D 60 C0        251          STA  DUMMY    ;
8BE4:88              252          DEY           ; ONE LESS LINE
8BE5:F0 9F     8B86  253          BEQ  BOTTOM   ; AT SCREEN BOTTOM?
8BE7:D0 D0     8BB9  254          BNE  NXTLN3   ;

8C00:          8C00  257         ORG  ST+$101  ; BURN BYTES TO MATCH VFFS

8C00:60 60 60 60     259 VPATRN DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8C10:60 60 60 60     261        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8C20:60 60 60 60     263        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8C30:60 60 60 60     265        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

3C40:60 60 60 60     267        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8C50:60 60 60 60     269        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8C60:60 60 60 60     271        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96


8C70:60 60 60 60     274        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8C80:60 60 60 60     276        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8C90:60 60 60 60     278        DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96
```

PROGRAM 13-2 cont

```
8CA0:60 60 60 60             DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8CB0:60 60 60 60             DFB 96,96,96,96,96,96,96,96,96,96,96,96,96,96,96,96

8CC0:60 60                   DFB 96,96


8CC2:                  ;         ** KEYPRESSED AND TIMEOUT **

8CC2:2C FB 8C          BTM      BIT   CONTROL   ; IS KEY EXIT ACTIVE?
8CC5:10 0A     8CD1             BPL   KEYOK     ;
8CC7:2C 00 C0                   BIT   KEYBD     ; LOOK FOR KEY
8CCA:10 05     8CD1             BPL   KEYOK     ;    NOT THERE?
8CCC:2C 10 C0                   BIT   KBSTRB    ; RESET KEYSTROBE
8CCF:38                QUIT     SEC             ; SET CARRY =
8CD0:60                         RTS             ;   EXIT ON KEY OR TIMEOUT

8CD1:EE FD 8C          KEYOK    INC   TIMEX     ; INCREMENT TIMEOUT MULTIPLIER
8CD4:2C FB 8C                   BIT   CONTROL   ; IS TIMER ACTIVE?
8CD7:50 0C     8CE5             BVC   NOQUIT    ;
8CD9:A9 1F                      LDA   #$1F      ; MASK FOR 1/32
8CDB:2D FD 8C                   AND   TIMEX     ;  AND TEST MULTIPLIER
8CDE:D0 05     8CE5             BNE   NOQUIT    ;
8CE0:CE FC 8C                   DEC   TIMER     ; ONE LESS COUNT
8CE3:F0 EA     8CCF             BEQ   QUIT      ;    DONE?
8CE5:18                NOQUIT   CLC             ; CLEARED CARRY =
8CE6:60                         RTS             ;   OK TO CONTINUE SCANNING


8CFB:          8CFB             ORG   BTM+$39   ; BURN BYTES TO MATCH VFFS

8CFB:C4                CONTROLDFB $C4         ; ARMS KP AND SETS TIMEOUT
8CFC:00                TIMER   DFB $00        ; COUNTER FOR TIMEOUT
8CFD:00                TIMEX   DFB $00        ; TIMEOUT * 32 MULTIPLIER


8CFE:AD FB 8C          SETUP    LDA   CONTROL   ; INITIALIZE TIMEOUT
8D01:29 3F                      AND   #$3F      ; MASK TIMEOUT BITS
8D03:8D FC 8C                   STA   TIMER     ;
8D06:60                         RTS             ; AND CONTINUE



8D07:                  ;          *** VAPORLOCK SUBROUTINE ***


8D07:                  ;  See the VAPORLOCK program for more use details.
8D07:                  ;

8D07:                  ;               ** SETUP **

8D07:EA                         NOP             ; EQUALIZE TO PAGE BOUNDARY
8D08:08                VAPORLK  PHP             ; SAVE FLAGS
8D09:48                         PHA             ; SAVE ACCUMULATOR

8D0A:A9 DB                      LDA   #ID0      ; WRITE ID PATCH
8D0C:8D F8 3F                   STA   $3FF8     ;  TO LINE #255
8D0F:8D F9 3F                   STA   $3FF9     ;
```

PROGRAM 13-2 cont

```
8D12:8D FA 3F                 STA  $3FFA    ;
8D15:8D FB 3F                 STA  $3FFB    ;
8D18:8D FC 3F                 STA  $3FFC    ;
8D1B:8D FD 3F                 STA  $3FFD    ;
8D1E:8D FE 3F                 STA  $3FFE    ;

8D21:A9 24                    LDA  #ID1     ;  TO LINE #256
8D23:8D F8 2B                 STA  $2BF8    ;
8D26:8D F9 2B                 STA  $2BF9    ;
8D29:8D FA 2B                 STA  $2BFA    ;
8D2C:8D FB 2B                 STA  $2BFB    ;
8D2F:A9 B6                    LDA  #ID2     ;
8D31:8D FC 2B                 STA  $2BFC    ;
8D34:8D FD 2B                 STA  $2BFD    ;
8D37:8D FE 2B                 STA  $2BFE    ;

8D3A:A9 49                    LDA  #ID3     ;  TO LINE 257
8D3C:8D F8 2F                 STA  $2FF8    ;
8D3F:8D F9 2F                 STA  $2FF9    ;
8D42:A9 88                    LDA  #ID4     ;
8D44:8D FA 2F                 STA  $2FFA    ;
8D47:8D FB 2F                 STA  $2FFB    ;

8D4A:A9 F3                    LDA  #ID5     ; AND FINALLY TO LINE 258
8D4C:8D F8 33                 STA  $33F8    ;
8D4F:A9 F2                    LDA  #ID5-1   ;
8D51:8D F9 33                 STA  $33F9    ;

8D54:A9 06                    LDA  #IDBYTE  ; CHECK FOR A IIe
8D56:CD B3 FB                 CMP  VERSION  ;
8D59:D0 05     8D60           BNE  MORE0    ;
8D5B:2C 19 C0       VBFIND    BIT  VBLANK   ; WAIT TILL IIe BLANKING START
8D5E:30 FB     8D5B           BMI  VBFIND   ;

8D60:2C 50 C0       MORE0     BIT  GR       ; FORCE FULL HIRES PAGE ONE
8D63:2C 57 C0                 BIT  HIRES    ;
8D66:2C 52 C0                 BIT  FULL     ;
8D69:2C 54 C0                 BIT  PAGE1    ; THEN FALL THROUGH TO LOCK


8D6C:               ;                 ** LOCK **


8D6C:A9 DB          LOCK      LDA  #ID0     ; SEARCH FOR FIRST PATCH ID VALUE
8D6E:CD 20 C0       RETRY1    CMP  SNIFF    ;
8D71:D0 FB     8D6E           BNE  RETRY1   ;
8D73:A9 02                    LDA  #02      ; DELAY FOR EXACTLY 57 CYCLES
8D75:20 A8 FC                 JSR  WAIT     ;   (HLINE-BNE-LDA#-LDA)
8D78:48                       PHA           ;
8D79:68                       PLA           ;
8D7A:AD 20 C0                 LDA  SNIFF    ; GET SECOND PATCH ID VALUE
8D7D:C9 B6                    CMP  #ID2     ; JITTER 4,5,OR 6?
8D7F:F0 06     8D87           BEQ  MORE1    ;
8D81:C9 24                    CMP  #ID1     ; JITTER 0,1,2, OR 3?
8D83:F0 02     8D87           BEQ  MORE1    ; OK TO CONTINUE
8D85:D0 E5     8D6C           BNE  LOCK     ; MISSED, TRY AGAIN
```

PROGRAM 13-2 cont

```
8D87:A9 02               MORE1   LDA   #02        ; DELAY EXACTLY 50 CLOCK CYCLES
8D89:20 A8 FC                    JSR   WAIT       ;  (HLINE-4-CMP#-BEQ-LDA#-LDA)
8D8C:AD 20 C0                    LDA   SNIFF      ; GET THIRD PATCH ID VALUE
8D8F:C9 88                       CMP   #ID4       ; JITTER 2 OR 3?
8D91:F0 04      8D97             BEQ   MORE2      ;
8D93:C9 49                       CMP   #ID3       ; JITTER 0 OR 1?
8D95:F0 02      8D99             BEQ   MORE3      ; ONLY WANT 2 CLOCK CORRECTION

8D97:D0 D3      8D6C     MORE2   BNE   LOCK       ; CURSES! FOILED AGAIN!
8D99:A9 02               MORE3   LDA   #$02       ; DELAY FOR EXACTLY 50 CYCLES
8D9B:20 A8 FC                    JSR   WAIT       ;  (HLINE-2-CMP-BEQ-BNE-LDA-LDA)
8D9E:AD 20 C0                    LDA   SNIFF      ; GET FOURTH PATCH ID VALUE
8DA1:4A                          LSR   A          ;   THEN SHIFT INTO CARRY
8DA2:B0 00      8DA4             BCS   MORE4      ;   TO EQUALIZE ONE COUNT

8DA4:C9 79               MORE4   CMP   #ID5/2     ; FINAL VALIDITY CHECK
8DA6:D0 C4      8D6C             BNE   LOCK       ; BACK TO SQUARE ONE
8DA8:EA                          NOP              ; HAVE LOCK AT THIS POINT


8DA9:                    ;                 ** STALL **


8DA9:A9 05               STALL   LDA   #$05       ; DELAY FOR EXACTLY 172 CYCLES
8DAB:20 A8 FC                    JSR   WAIT       ;
8DAE:A9 01                       LDA   #$01       ;
8DB0:20 A8 FC                    JSR   WAIT       ;

8DB3:A9 06               FIX2+   LDA   #IDBYTE    ; ADD ONE EXTRA CYCLE ONLY ON
8DB5:CD B3 FB                    CMP   VERSION    ;  THE II+ TO EQUALIZE ON-SCREEN
8DB8:D0 00      8DBA             BNE   SHOW       ;  DISPLAY MODE SWITCHING

8DBA:2C 20 C0            SHOW    BIT   SNIFF      ; OPTIONAL MODE CHANGES GO HERE
8DBD:2C 20 C0                    BIT   SNIFF      ;
8DC0:2C 20 C0                    BIT   SNIFF      ;
8DC3:2C 20 C0                    BIT   SNIFF      ;

8DC6:18                  PHASE   CLC              ; PHASING CHANGES GO HERE
8DC7:90 00      8DC9             BCC   MORE5      ;
8DC9:EA                  MORE5   NOP              ; BCC TAKEN = 2; NOT TAKEN = 3
8DCA:EA                          NOP              ; NOP = 2 CHARACTERS

8DCB:68                          PLA              ; RESTORE ACCUMULATOR AND FLAGS
8DCC:28                          PLP              ;
8DCD:60                          RTS              ; AND EXIT
```

PROGRAM 13-3 An Applesoft program to upgrade older VFFS files into the new VLFFS format.

```
10  REM  ********************
12  REM  *                  *
14  REM  *   VFFS TO VLFFS  *
16  REM  *  FILE  CONVERTER *
18  REM  *                  *
20  REM  *    VERSION 1.0   *
21  REM  * APPLE  II+ OR IIE *
22  REM  *     (9-25-83)    *
23  REM  *                  *
24  REM  *   COPYRIGHT 1983 *
26  REM  * BY  DON LANCASTER *
28  REM  *  AND SYNERGETICS *
30  REM  *                  *
32  REM  *     BOX 1300     *
34  REM  * THATCHER AZ 85552 *
36  REM  *  (602)  428-4073 *
38  REM  *                  *
40  REM  *  ALL  COMMERCIAL *
42  REM  *  RIGHTS RESERVED *
44  REM  *                  *
46  REM  ******************** [J]
 [J]

100  HIMEM: 30000: TEXT : HOME : CLEAR [J]

110  PRINT "    VFFS TO VLFFS FILE CONVERTER     "
120  PRINT "...................................."
130  PRINT : PRINT "ONE OR TWO DRIVES ..... ";:
     GET D$: PRINT D$
140  PRINT : INPUT "OLD VFFS FILENAME ..... ";A$
150  PRINT : INPUT "NEW VLFFS FILENAME? ... ";B$
160  PRINT : PRINT : PRINT : PRINT : PRINT
     "OK TO CONTINUE (Y/N)? ";: GET Z$ [J]

170  IF Z$ >< "Y" THEN 100 [J]

180  C$ = "VLFFS.EMPTY"
190  POKE 34,2: REM  SAVE HEADER
200  HOME : PRINT : PRINT : PRINT
     "PUT ";C$;" INTO DRIVE ";D$: PRINT : PRINT
     "THEN HIT ANY KEY TO CONTINUE ";: GET Z$
210  HOME : PRINT : PRINT : PRINT "I AM LOADING ";C$;:
     PRINT " FROM DRIVE ";D$
220  PRINT "[D]BLOAD ";C$;",D";D$
300  HOME : PRINT : PRINT : PRINT
     "PUT ";A$;" INTO DRIVE 1": PRINT : PRINT
     "THEN HIT ANY KEY TO CONTINUE ";: GET Z$
310  HOME : PRINT : PRINT : PRINT
     "I AM LOADING ";A$;" FROM DRIVE 1"
320  PRINT "[D]BLOAD";A$;",A$87FF,D1"
```

PROGRAM 13-3 cont

```
400  HOME : PRINT : PRINT : PRINT
     "I AM MOVING DATA VALUES" [J]

410  H(1) = 34865:H(2) = 34964:H(3) = 35012:H(4) = 34913
420  FOR KK = 0 TO 4: FOR LL = 0 TO 10:
430  POKE (H(KK) + 3 * LL + 768), PEEK (H(KK) + 3 * LL)
440  ZZ =  PEEK (49200): NEXT LL,KK
500  VP = 35072
510  FOR LL = 0 TO 192
520  POKE (VP + LL + 768), PEEK (VP + LL)
530  ZZ =  PEEK (49200): NEXT LL
600  POKE 36091, PEEK (35323) [J]

620  HOME : PRINT : PRINT : PRINT
     "PUT THE DISK YOU WANT ";B$;" ON";
630  PRINT : PRINT : PRINT "INTO DRIVE ";D$;"."
640  PRINT : PRINT "THEN HIT ANY KEY TO CONTINUE ";:
     GET Z$
650  PRINT  [J]

700  HOME : PRINT : PRINT : PRINT
     "I AM NOW SAVING ";B$;" TO DRIVE ";D$
710  PRINT "[D]BSAVE";B$;",A$8AFF,L$2CF,D";D$
720  PRINT [J]

800  HOME : PRINT : PRINT : PRINT : PRINT
     "FINISHED.  SHALL WE DO ANOTHER (Y/N) ": GET Z$
810  IF Z$ = "Y" THEN 100
820  TEXT : HOME : CLEAR : END
```

THE II+ AND OTHER GOTCHAS

The Vaporlock and VLFFS.EMPTY work best on the Apple IIe. There are some limitations and compromises you should know about if you want both of these to work properly on both the Apple II+ and Apple IIe.

First, the II+ switches its screen modes one cycle earlier than the IIe. An automatic compensator is built into the Vaporlock at FIX II+ that will make both types of Apples switch at the same screen point. For some oddball, nonscreen field switching on the II+, you may want to defeat this compensation.

Secondly, there will be some glitches in on-screen changes made by the II+. The GLITCH STOMPER hardware mod of Enhancement 6 minimizes, but does not eliminate these. Glitches are far simpler to manage on the IIe. Just exit from text on a space and exit from LORES black, and all IIe glitches should be invisible.

Thirdly, some rare plug-ins on the II+ may defeat the Vaporlock by loading the data bus too heavily. The worst culprits seem to be any LSTTL gates that directly connect to the data bus and pull up any floating zeros.

Send us a list of any problem boards you find.

Problem boards won't stall the Vaporlock on the IIe, since IIe slots 1–7 are fully buffered. Just about anything plugged into slot 0 of the IIe will terminate the data bus in a suitable transceiver, so there should be no IIe hassles with any known plug-in. The IIc, of course, has none of these problems.

Fourthly, a brief flash of the contents of HIRES page one will whip by on the first Vaporlock locking on the II+. This is automatically eliminated in the IIe by use of the $C019 blanking signal. For this II+ flash to be invisible, all of the contents of HIRES full page one should look similar to what was previously displayed, what will be displayed, or else should be left all black.

The bottom line is that you can do more and do it better by limiting commercial uses of the Vaporlock to the IIe, although it is definitely possible to do unique and interesting stuff in mixed fields that run on any "real" Apple.

Please note that the Vaporlock and VLFFS.EMPTY will *not* run on a Franklin, and may not run on a clone. The Franklin uses only 64 clock cycles per line, compared to Apple's 65.

What the vaporlock will do on a clone is anyone's guess.

Better repeat that . . .

> The Vaporlock will only run for sure on "real" Apples!
>
> Franklins and clones need not apply.

You can relocate Vaporlock in any protected space you want so long as you do not cross any page boundaries after the relocation. The Vaporlock is fully relocatable object code; all you have to do is move it. Any VLFFS files are best moved by reassembly to a new origin.

As a final gotcha, remember that any text mode that exits during the horizontal blanking time will try to kill the color. If too many horizontal lines do not have color reference bursts, your color monitor or tv might get its colors mixed up or dropped entirely. Fortunately, it takes lots of missed bursts to foul up the works on most sets, so just keep as many lines out of the text mode during *horizontal* blanking as you can.

AN OFF-THE-WALL VAPORLOCK USE

Chart 13-2 shows a hex dump of a module I call VLFFS.NOCOLOR. When you BRUN this module, any color HIRES display will be shown in black and white only. Color returns on any key being pressed. Such a color killer is most useful when showing HIRES graphs or business graphics without any annoying color fringes.

If you don't like hand loading, the program is available and ready to BRUN on the companion diskette.

Naturally, everyone "knows" that it is absolutely impossible to build a software-only color killer that works equally well on a II+, IIc, or IIe. Only nobody bothered to tell VLFFS.NOCOLOR.

I'll leave this one for you to puzzle over. You'll need a thorough understanding of VLFFS.EMPTY, of exact field sync, of the color killer circuit and of Enhancement 3's tearing method to figure it out. I've purposely left some slop in the program so you can shorten it and further improve it.

Sneaky, huh?

Chart 13-2 Hex Dump of VLFFS.NOCOLOR

```
8AFF- EA

8B00- 20 FE 8C 20 21 8B 20 0E
8B08- 8B 2C 50 C0 90 F5 60 B6
8B10- D2 FF A0 80 A0 D2 A0 A0
8B18- A0 A0 A0 A0 A0 A0 A0 B1
8B20- D0 20 08 8D A0 FC AD 00
8B28- 8C 30 63 10 00 29 7F AA
8B30- 9D 00 C0 8D 60 C0 8D 60
8B38- C0 8D 60 C0 8D 60 C0 8D

8B40- 60 C0 8D 60 C0 8D 60 C0
8B48- 8D 60 C0 8D 60 C0 8D 51
8B50- C0 88 F0 32 D0 D0 B9 00
8B58- 8C 30 D0 10 00 29 7F AA
8B60- 9D 00 C0 8D 60 C0 8D 60
8B68- C0 8D 60 C0 8D 60 C0 8D
8B70- 60 C0 8D 60 C0 8D 60 C0
8B78- 8D 60 C0 8D 60 C0 8D 60

8B80- C0 88 F0 02 D0 D0 4C C2
8B88- 8C B9 00 8C 30 30 10 00
8B90- 29 7F AA 9D 00 C0 8D 60
8B98- C0 8D 60 C0 8D 60 C0 8D
8BA0- 60 C0 8D 60 C0 8D 60 C0
8BA8- 8D 60 C0 8D 60 C0 8D 60
8BB0- C0 8D 60 C0 88 F0 CF D0
8BB8- D0 B9 00 8C 30 9D 10 00

8BC0- 29 7F AA 9D 00 C0 8D 60
8BC8- C0 8D 60 C0 8D 60 C0 8D
8BD0- 60 C0 8D 60 C0 8D 60 C0
8BD8- 8D 60 C0 8D 60 C0 8D 60
8BE0- C0 8D 60 C0 88 F0 9F D0
8BE8- D0 A0 B0 A0 A0 A0 D4 A0
8BF0- A0 A0 A0 A0 A0 A0 A0 A0
8BF8- A0 A0 FF A0 A0 A0 A0 A0

8C00- 50 50 50 50 50 50 50 50
8C08- 50 50 50 50 50 50 50 50
8C10- 50 50 50 50 50 50 50 50
8C18- 50 50 50 50 50 50 50 50
8C20- 3E 50 50 50 50 50 50 50
8C28- 50 50 50 50 50 50 50 50
8C30- 50 50 50 50 48 50 50 50
8C38- 50 50 50 50 50 50 50 50

8C40- 50 50 50 50 50 50 50 50
8C48- 50 50 50 50 50 50 50 50
8C50- 4E 50 50 50 50 50 50 50
8C58- 50 50 50 50 50 50 50 50
8C60- 50 50 50 50 50 50 50 50
8C68- 50 50 50 50 50 50 50 50
8C70- E9 50 50 50 50 50 50 50
8C78- 50 50 50 50 50 50 50 F7

8C80- 50 50 50 50 50 50 50 50
8C88- 50 50 50 50 50 50 50 50
8C90- 50 50 50 50 50 50 50 50
8C98- 50 50 50 50 50 50 50 50
8CA0- 50 50 50 50 50 50 50 50
8CA8- 50 50 50 50 50 50 50 50
8CB0- 50 50 50 50 4F 50 50 50
8CB8- 50 50 50 50 50 50 50 50

8CC0- 50 50 2C FB 8C 10 0A 2C
8CC8- 00 C0 10 05 2C 10 C0 38
8CD0- 60 EE FD 8C 2C FB 8C 50
8CD8- 0C A9 1F 2D FD 8C D0 05
8CE0- CE FC 8C F0 EA 18 60 A0
8CE8- C9 B4 A0 A0 C4 FF A0 A0
8CF0- A0 A0 A0 D4 B8 A0 A0 B8
8CF8- A0 A0 A0 80 00 79 AD FB

8D00- 8C 29 3F 8D FC 8C 60 EA
8D08- 08 48 A9 DB 8D F8 3F 8D
8D10- F9 3F 8D FA 3F 8D FB 3F
8D18- 8D FC 3F 8D FD 3F 8D FE
8D20- 3F A9 24 8D F8 2B 8D F9
8D28- 2B 8D FA 2B 8D FB 2B A9
8D30- B6 8D FC 2B 8D FD 2B 8D
8D38- FE 2B A9 49 8D F8 2F 8D

8D40- F9 2F A9 88 8D FA 2F 8D
8D48- FB 2F A9 F3 8D F8 33 A9
8D50- F2 8D F9 33 A9 06 CD B3
8D58- FB D0 05 2C 19 C0 30 FB
8D60- 2C 50 C0 2C 57 C0 2C 52
8D68- C0 2C 54 C0 A9 DB CD 20
8D70- C0 D0 FB A9 02 20 A8 FC
8D78- 48 68 AD 20 C0 C9 B6 F0

8D80- 06 C9 24 F0 02 D0 E5 A9
8D88- 02 20 A8 FC AD 20 C0 C9
8D90- 88 F0 04 C9 49 F0 02 D0
8D98- D3 A9 02 20 A8 FC AD 20
8DA0- C0 4A B0 00 C9 79 D0 C4
8DA8- EA A9 05 20 A8 FC A9 01
8DB0- 20 A8 FC A9 06 CD B3 FB
8DB8- D0 00 2C 20 C0 2C 20 C0

8DC0- 2C 20 C0 2C 20 C0 18 90
8DC8- 00 EA EA 68 28 EA EA 60
```

This software-only color killer is available ready to run on the companion diskette to this volume.

SO WHAT GOOD IS AN EXACT LOCK?

What good is all this? Now you can mix HIRES, LORES, and text all over the screen. But what can you *really* do with mixed fields that was impossible or very hard before?

Let's wrap up this enhancement with 13 brand new uses for exact field sync.

1. There must be something to **subliminal ad messages** because they are an absolutely illegal no-no for commercial uses. How can you flash a subliminal message for one field at full intensity and then for a second field at half intensity, inserted into "normal" program material?
2. How can you include **crosshairs** into a HIRES display that does not change or alter the database you are looking at? What uses can you think of for being able to identify and then analyze a single horizontal line of video?
3. How much **grey scale** can you get out of your Apple? You can get one grey level by alternating fields. Stuff on both fields will be white. Stuff on one field will be grey. Stuff on no fields will be black. But, what if you add hardware between the annunciator outputs and the video combiner circuitry so that the white level changes with the annunciator values?
4. Once you have grey scale, how do you use **anti-aliasing** to get rid of the "jaggies" on slanted lines? The big animation boys don't have any jaggies, and they don't need lots of extra resolution to get rid of them.
5. Dynamically changing the switching positions in a mixed-field display would reveal different portions of different screen parts as time progressed. Shows how you can do **dynamic video wipes** to handle cuts, wipes, and fades that rival commercial tv.
6. Snapping off-screen photographs of an Apple display is no trivial matter. Show an **electronic shutter control** that exposes film for exactly one whole field of video, no more, no less. Arrange your electronic and mechanical time delays so the shutter opens during one vertical blanking and closes during the next one.
7. How about a **tail-less light pen** small enough to wear on a finger without being unpleasant or interfering with typing. Use a level-switched button cell battery or else a solar cell for power. Mixed fields could tell where you were on the screen and dramatically simplify the response circuitry. The part worn would communicate to the Apple with ultrasonics or infrared with simple pulses.
8. What is possible in the way of **text over color** and mixed color displays? How can you minimize flicker? The new, double resolution LORES and HIRES on the IIe should be exploitable here. Can you legibly "float" text over an adventure display? What can you do that really looks great along these lines?
9. How about some **odometer** or **gas pump** style displays where inset numbers smoothly rotate, rather than jumping from one number to another? Or slot machines with smoothly rotating wheels? Bomber flybys? How can mixed fields help you here?
10. Show how to insert a **real-time clock** or other score timing device into the top line of a HIRES display. Can you do this without using interrupts? Without needing hex-to-decimal conversions? Without needing any hardware mods at all?
11. Can **classical cell-by-cell animation** be done on an Apple? Can you plot only screen changes from cell to cell? Can you do this "free form" without byte boundary limits? How can mixed fields help here?
12. What **new uses for joysticks, mice, or paddles** can you think of in which these directly control the points at which field switching takes place on your Apple? There should now be time enough to read a paddle between field lockings. What else can your joystick control?

13. There are lots of **new field switches** on the Apple IIe. What happens if you start flipping these in sync to the screen in new, different, and unusual ways? Here is where the real mind-blowers will show up. What can you think of?

And those should just about be enough to get you started. Use the response card in the back of the book to tell us what you come up with and to ask any "what if?" or "why not?" style exact field sync questions.

The following programs are included in the companion diskette to this volume:

VAPORLOCK.SOURCE
VAPORLOCK
VLFFS.SOURCE
VLFFS
VFFS>VLFFS CONVERTER

VLTEST
ALTERNATER
VLFFS.EMPTY
VLFFS.WINDOW
VLFFS.NOCOLOR

FUN WITH VLFFS FIELDS
VLFFS.BOXES
VLFFS.GRAPH
VLFFS.GIRLS
VLFFS.BYE

See the cards in back of the book for feedback, hotline, and ordering information.

Apple Enhancer Support Services

Don Lancaster and Synergetics offer many different support services for Apple enhancers and others wishing to push the limits of their Apple capabilities.

Included now in these services are feedback cards, the "13-day" pre-release service, companion support diskettes, the Gila Valley Apple Growers Association, and a no-charge voice hotline.

FEEDBACK RESPONSE CARDS

The feedback response cards at the back of this book are your way of closing the loop and letting us know about any problems you may have, or to tell us what you want to see in the way of future enhancements. Simply tear out the response card, fill it out, and drop it in the mail.

Keep us posted on your progress and exploration of all of the enhancements.

THE 13-DAY PRE-RELEASE SERVICE

Just as soon as new Don Lancaster individual packages become available, you will be able to get them on a "preview" basis, long ahead of when they can appear in print. These materials will always be preliminary and will have no fancy packaging or elaborate distribution.

To stay up to date on the "13-day" program, just use the response card to show us your interest, or else watch for small ads appearing in *Apple Assembly Line, Call A.P.P.L.E., Modern Electronics,* or *Computer Shopper.*

COMPANION DISKETTES

A Volume 2 companion diskette is available that holds all of the code shown you in Enhancements 9 through 13, along with a few bonus programs.

Here's a list of the . . .

Partial Contents of the Companion Diskette for Enhance Volume 2

DGLOSS
EGLOSS
AWIIE NULLIFIER
AWIIE STRETCHIFIER
WPL.FORMAT DIABLO 630

WPL.FORMAT D630 NOFRILLS
WPL.DETAIL VAPORLOCK
WPL.DETAIL E9
WPL.DETAIL ALL
WPL.CAMERA READY

WPL.BULLET SHOOTER
ART EXAMPLES D630
SNATCHMON
KREBFSPELL.SOURCE
KREBFSPELL

SNEAKYSTUFF
CURSDSTRING
TRIPLE DELAY FINDER
VAPORLOCK.SOURCE
VAPORLOCK

VLFFS.SOURCE
VLFFS
VFFS>VLFFS CONVERTER
VLTEST
ALTERNATER

VLFFS.EMPTY
VLFFS.WINDOW
VLFFS.NOCOLOR
FUN WITH VLFFS FIELDS
VLFFS.BOXES

VLFFS.GRAPH
VLFFS.GIRLS
VLFFS.BYE

. . . plus a surprise or two

This 33+ program DOS 3.3e diskette costs only $19.50. It is fully copyable for your personal use only and includes complete EDASM source code listings.

You can order this support diskette using the order card bound in the back of the book. Visa® and MasterCard® are accepted, as are telephone orders via the hotline.

In addition, all of Enhancements 9 and 12, plus bunches more, are available as the eight diskette side AWIIe TOOLKIT package. These give you all the charts, text, and tables in "machine readable" form for serious or advanced users.

On any Don Lancaster or Synergetics service, please note that purchase orders and CODs can NOT be accepted. Orders to a foreign address or in foreign funds are also NOT acceptable.

A number of other books and diskette packages are also available, including the support diskettes for Enhance I and Assembly I, the AWIIe TOOLKIT, the ProDOS Apple Writer 2.0 TOOLKIT, and *The Incredible Secret Money Machine*, Don's underground classic book on forming your own winning technical ventures. Also available are some exciting new graphics animation demos. See the order card for full details.

THE GILA VALLEY APPLE GROWERS ASSOCIATION

The Gila Valley Apple Growers Association is a most unusual consortium of Apple owners and users. The meetings are a "test bed" for evaluation of new software products in development by its members. New ways of "pushing the limits" of Apple and other hardware and software are the usual activity focus. There is also a special interest group on tinaja questing.

Membership is more or less free, but is strictly and exclusively limited to those attending the meetings. There are not, and never will be, any printed notices, newsletters, or outside library exchange services available.

By the way, you pronounce the "G" as if it were an "H", like in "Hee-luh".

The association meets 6–10 PM every Wednesday night during the school year here in Thatcher, usually in Eastern Arizona College room T8 or T9.

Stop in sometime.

THE APPLE ENHANCER'S HOTLINE

A voice hotline service is available as a joint service of Synergetics, Don Lancaster, and the Gila Valley Apple Growers Association . . .

Apple Enhancers Voice Hotline
(602) 428-4073

The service is free, except for your phone charges. Best time to call is 8–5 weekdays, Mountain Standard Time. The two main areas of expertise include Apple Writer IIe and assembly language programming.

CARDS MISSING?

If the cards are missing, call or write . . .

SYNERGETICS
746 First Street
Box 809
Thatcher, AZ 85552

(602) 428-4073

INDEX

M•O•R•E ◊ F•R•O•M ◊ S•A•M•S

☐ **ENHANCING YOUR APPLE II, Volume 1 (2nd Edition)**
Don Lancaster shows you how to mix text, low-res and high-res anywhere on-screen; do 3-D graphics, overlapping single-line colors and other special effects, have gentle scrolls, generate 191 background colors and quickly analyze any machine-language program. Don Lancaster.
ISBN 0-672-21822-4 . . . . **$15.95**

☐ **ASSEMBLY COOKBOOK FOR THE APPLE II/IIe**
Strong Lancaster seminar on assemblers and how to use them, plus step-by-step instruction leading you through practical modules of working assembly language code. Excellent stuff! Don Lancaster.
ISBN 0-672-22331-7 . . . . **$21.95**

☐ **DON LANCASTER'S MICRO COOKBOOK, Volume 1**
Down-to-earth coverage of all microcomputer and microprocessor fundamentals needed to start understanding machine language programming. Can be applied to any micro. Don Lancaster.
ISBN 0-672-21828-3 . . . . **$15.95**

☐ **DON LANCASTER'S MICRO COOKBOOK, Volume 2**
Teaches you machine-language programming on the microprocessor family and microcomputer of your choice, using Don's "those #$!&#%$! cards" technique. Covers address space, addressing, system architecture, I/O. Don Lancaster.
ISBN 0-672-21829-1 . . . . **$15.95**

☐ **TV TYPEWRITER COOKBOOK**
Heaven for hobbyists. Discusses TVT comunications, basic TVT system design, memory types, interface circuitry, hard copy output and color graphics. Don Lancaster.
ISBN 0-672-21313-3 . . . . **$12.95**

☐ **THE CHEAP VIDEO COOKBOOK**
Logical sequel to *TV Typewriter Cookbook.* Demonstrates methods for hobbyists to use in getting words, pictures, and code from a computer to a TV with no electronic modifications required. Don Lancaster.
ISBN 0-672-21524-1 . . . . **$8.95**

☐ **SON OF CHEAP VIDEO**
This sequel to *Cheap Video Cookbook* makes cheap video even cheaper with transparency detail for less than $1 plus details for a complete $7 video display system. Don Lancaster.
ISBN 0-672-21723-6 . . . . **$10.95**

☐ **THE HEXADECIMAL CHRONICLES**
This bookful of instant, hassle-free tables lets you do 52 computing calculations and conversions easily. Don Lancaster.
ISBN 0-672-21802-X **$17.95**

☐ **TTL COOKBOOK**
A complete look at TTL, including what it is, how it works, how it's interconnected, how it's powered and how it's used in many practical applications. Don Lancaster.
ISBN 0-672-21035-5 . . . . **$12.95**

☐ **CMOS COOKBOOK**
Explains in cookbook format how CMOS differs from other MOS designs, how it's powered and what its advantages are over other constructions. Includes circuits you can build. Don Lancaster.
ISBN 0-672-21398-2 . . . . **$14.95**

☐ **ACTIVE-FILTER COOKBOOK**
Don Lancaster presents a menu of pre-designed filters you can borrow and adapt when you need an active filter but don't want to take the time to design it. Don Lancaster.
ISBN 0-672-21168-8 . . . . **$14.95**

☐ **INTRODUCING THE APPLE® IIc**
Indispensable introductory guide describes all IIc features, discusses compatibility and differences with the IIe and covers IIc setup, expansion, graphics and communications. Philip Lieberman.
ISBN 0-672-22393-7 . . . . **$17.95**

☐ **INTRODUCING THE APPLE MACINTOSH™**
Explores Macintosh design philosophy, physical structure, displays, keyboard, mouse, software, and accessories. Covers graphics, word processing, spreadsheets, BASIC and windowing. Connolly and Lieberman.
ISBN 0-672-22361-9 . . . . **$12.95**

☐ **MACINTOSH USER'S GUIDE**
Is the Macintosh right for you? Compares the Mac with 5 other best-selling micros and then explains fundamental and advanced applications. Gordon McComb.
ISBN 0-672-22328-7 . . . . **$16.95**

☐ **APPLESOFT FOR THE IIe**
A detailed Applesoft programmer's reference manual written specifically for the Apple IIe and covering all aspects of IIe syntax and programming techniques. Blackwood and Blackwood.
ISBN 0-672-22259-0 . . . . **$19.95**

☐ **BASIC TRICKS FOR THE APPLE**
Filled with ideas and examples for input routines, rounding, report formatting, working with dates and times, and sorting. Do your own loading or save typing time with the disk-based Combo Pack. Allen L. Wyatt.
Book: ISBN 0-672-22208-6 . . . . **$8.95**
Combo Pack: ISBN 0-672-26225-8 . . . . **$24.95**

☐ **INTIMATE INSTRUCTIONS IN INTEGER BASIC**
Integer BASIC executes more quickly than Applesoft. Learn to build Integer programs which run smoothly and take full advantage of that dialect's speed. Blackwood and Blackwood.
ISBN 0-672-21812-7 . . . . **$8.95**

☐ **MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE II**
Book 1—Over 30 BASIC programs to help you save money on energy usage, make bar charts, dial your telephone or learn a foreign language. Also includes an electronic harpsichord, a tarot card reader and some two-level dungeons. Howard Berenbon.
ISBN 0-672-21789-9 . . . . **$13.95**

Book 2—Continues with dungeons, educational programs, budget analysis, a weekly calendar, a series on money and investment and programs on ESP. Howard Berenbon.
ISBN 0-672-21864-X **$12.95**

M•O•R•E ◊ F•R•O•M ◊ S•A•M•S

☐ APPLESOFT LANGUAGE (2nd Edition)

Valuable as a tutorial or for reference on disk operation, syntax and programming in Applesoft, advanced programming techniques, graphics, color commands, sorts and searches. Blackwood and Blackwood.
ISBN 0-672-22073-3 . . . . **$14.95**

☐ APPLE LOGO PROGRAMMING PRIMER

Quickly learn and use the complete Logo language. Emphasizes top-down programming and covers recursion, outputs and utilities. Includes clear and concise diagram explanations of Logo syntax and many sample programs. Martin, Prata and Paulsen.
ISBN 0-672-22342-2 . . . . **$19.95**

☐ 88 APPLE LOGO PROGRAMS

Eighty-eight fully tested and ready-to-run Logo programs, including database and graphing packages for home and business as well as entertainment, special turtle graphics programs and a powerful "Matchmaker" program. Waite, Martin, and Martin.
Book: ISBN 0-672-22343-0 . . . . **$15.95**
Combo Pack: ISBN 0-672-26224-X **$29.95**

☐ KID-POWERED LOGO

The fictional character, Mr. Graphix, joins with his new friend Mr. Thinker, to provide children with an introduction to Logo. Uses many illustrations, large type and simple language. David J. Fiday.
ISBN 0-672-22190-X **$17.95**

☐ APPLE GAMES

Learn how to use text, graphics and sounds in developing your own computer games. Complete program listings are provided for every program. Do your own loading or save time and effort with the disk-based Combo Pack. Allen L. Wyatt.
Book: ISBN 0-672-22394-5 . . . . **$8.95**
Combo Pack: ISBN 0-672-26226-6 . . . . **$24.95**

☐ APPLE II FOR KIDS FROM 8 TO 80

Large format, varied activities, conversational approach and extensive graphics all combine with special computer camp principles to help you learn fast. Zabinski and Mazzola.
ISBN 0-672-22297-3 . . . . **$10.95**

☐ KID-POWERED GRAPHICS

This simple, direct and jargon-free treatment focuses on commands, variables and program planning for students and teachers in grades 3-8. David J. Fiday.
ISBN 0-672-22229-9 . . . . **$14.95**

☐ POLISHING YOUR APPLE, Volume 1

End your search through endless manuals with this clearly written, highly practical, concise assemblage of the procedures needed for writing, filing and printing programs. Herbert M. Honig.
ISBN 0-672-22026-1 . . . . **$4.95**

☐ POLISHING YOUR APPLE, Volume 2

Create attractive menus, do effective error-trapping and improve program flow with routines to format currency entries, eliminate using return and generate 4-digit day, month, year dating. Herbert M. Honig.
ISBN 0-672-22160-8 . . . . **$4.95**

☐ APPLE PROGRAMMER'S HANDBOOK

Just the thing to get you started in assembly language on the Apple. Packed with essential Apple data, dozens of tested "stock" routines organized by topics and a detailed memory map. Paul Irwin.
ISBN 0-672-22175-6 . . . . **$22.95**

☐ APPLE IIe PROGRAMMER'S REFERENCE GUIDE

Makes needed IIe facts, applications and other technical information readily available at your fingertips. David L. Heiserman.
ISBN 0-672-22299-X **$19.95**

☐ APPLE II ASSEMBLY LANGUAGE

Specifically designed to show beginning assembly language programmers how to use the Apple's three character, 56-word assembly vocabulary to create powerful, fast-running programs. Marvin L. DeJong.
ISBN 0-672-21894-1 . . . . **$15.95**

☐ INTERMEDIATE LEVEL APPLE II HANDBOOK

Practical information that uses ROM-based Integer BASIC to lead you into Apple 6502 machine and assembly language programming. Learn to mix machine code with BASIC and use Integer's miniassembler, too. David L. Heiserman.
ISBN 0-672-21889-5 . . . . **$16.95**

☐ APPLE FORTRAN

Only fully detailed Apple FORTRAN manual on the market for beginning and advanced programmers, businessmen and other professionals. Blackwood and Blackwood.
ISBN 0-672-21911-5 . . . . **$14.95**

☐ CIRCUIT DESIGN PROGRAMS FOR THE APPLE II

Applesoft programs to show you "what happens if" and "what's needed when" as they apply to periodic waveform, rms and average values and the solution of simultaneous equations. Howard M. Berlin.
ISBN 0-672-21863-1 . . . . **$15.95**

☐ THE APPLE II CIRCUIT DESCRIPTION

Provides a detailed circuit description of the Apple II motherboard, keyboard and power supply. Covers all Apple II and II+ revisions and includes timing diagrams for most signals. Winston Gayler.
ISBN 0-672-21959-X **$22.95**

☐ APPLE II PLUS/IIe TROUBLESHOOTING AND REPAIR GUIDE

Troubleshooting flowcharts that allow you to diagnose and remedy the probable cause of failure, plus advanced troubleshooting for more complicated repairs. Robert C. Brenner.
ISBN 0-672-22353-8 . . . . **$19.95**

☐ DISKS, FILES, AND PRINTERS FOR THE APPLE II

Easy-to-follow instructions for using disks and printers with the Apple II, plus tips on programming with sequential, random and EXECutive files also included. Blackwood and Blackwood.
ISBN 0-672-22163-2 . . . . **$15.95**

M•O•R•E ◊ F•R•O•M ◊ S•A•M•S

☐ APPLE II APPLICATIONS

Presents a broad spectrum of tested programming and board-level interfacing applications, including serial and parallel I/O boards, EPROM or E2PROM boards and remote data acquisition. Marvin L. DeJong.
ISBN 0-672-22035-0 .... **$13.95**

☐ APPLE INTERFACING

Allows engineers, technicians, and hobbyists to master the important task of successfully interfacing an Apple computer with a variety of peripheral devices. Titus, Larsen and Titus.
ISBN 0-672-21862-3 .... **$11.95**

☐ FINANCIAL PLANNING MIND TOOLS

Seventeen pre-formed model overlays, or templates, allow you to quickly use your spreadsheet for virtually every financial calculation you will ever need. Expert Systems, Inc. Apple II versions requires Apple II compatible system, 64K RAM, one disk drive and Multiplan or VisiCalc software.
For Multiplan
ISBN 0-672-29058-8 .... **$79.95**
For VisiCalc
ISBN 0-672-29059-6 .... **$79.95**

☐ MONEY TOOL

Keeps accurate records of income and expenses, balances and reconciles your checkbook and helps manage your bank accounts for maximum interest. Herb Honig. Requires Apple II compatible system, 48K RAM, DOS 3.3, Applesoft in ROM, one disk drive and printer.
ISBN 0-672-26235-5 .... **$29.95**

☐ PEN PAL

Affordable and easy to use word processing for your Apple. Left/right justifies, centers, indents, auto page-numbers. Does text search, search and replace, delete and block moves. Many on-line help menus. Moller and Moller. Requires Apple II compatible system with 48K RAM, DOS 3.3, one disk drive and printer.
ISBN 0-672-26234-7 .... **$29.95**

☐ INSTANT RECALL

Extremely fast, memory-based freeform data base allows instantaneous search and retrieval of characters, words or combinations thereof. Supports all types of printers. Charles R. Landers. Requires Apple II compatible system, 48K RAM, DOS 3.3, Applesoft in ROM and one disk drive. 80-column printer optional.
ISBN 0-672-26233-9 .... **$29.95**

☐ HELLO CENTRAL!

Terminal program lets you dial voice calls, take messages and use the built-in text editor to prepare, save, retrieve, manipulate and print your communications. Bruce Kallick. Requires Apple II compatible system, 48K RAM, DOS 3.3, one disk drive and modem. Printer and hard disk system optional.
ISBN 0-672-26081-6 .... **$99.95**

☐ COMPUTER GRAPHICS USERS GUIDE

Subjects include basic geometry, fundamental computing, turning your ideas into pictures and transferring these pictures from the computer to video tape or film. Many beautiful, full-color photographs. Andrew S. Glassner.
ISBN 0-672-22064-4 .... **$19.95**

☐ KEYFAX FOR THE APPLE II/IIe

Two-color, rugged plastic templates fit around your keyboard and provide required keystroke information for DOS and applications software. Howard W. Sams & Co., Inc.

☐ VisiCalc/DOS 3.3

No. 26152 .... **$12.95**

☐ Multiplan/DOS 3.3

No. 26173 .... **$12.95**

☐ Applewriter IIe/DOS 3.3

No. 26200 .... **$12.95**

☐ ProDOS (both sides)

No. 26202 .... **$12.95**

☐ Applesoft BASIC (both sides)

No. 26208 .... **$12.95**

☐ THE CAD/CAM PRIMER

Learn the basics about CAD/CAM and its probable impact on our lives. Daniel Bowman.
ISBN 0-672-22187-X **$14.95**

☐ APE ESCAPE

As Harry the Ape, your task is to climb a skyscraper while avoiding bowling balls, falling nets, a zookeeper on a scaffold and earthquakes. Kevin Bagley. Requires Apple II compatible system, 48K RAM, DOS 3.3, Applesoft in ROM, one disk drive.
ISBN 0-672-26242-8 .... **$19.95**

☐ REGATTA

Complete documentation allows even a novice to quickly enjoy this accurate simulation of sailing on a small but demanding scale. DeMuth and Peterson. Requires Apple II compatible system, 48K RAM, DOS 3.3, one disk drive.
ISBN 0-672-26237-1 .... **$19.95**

☐ SPUD/MUG SHOT

An incredible, fast-action, challenging game in which you and your opponent each attempt to deflect the spud toward the other's fort. Stephen Walloch. Requires Apple II compatible system, 48K RAM, DOS 3.3, Applesoft in ROM, one disk drive. Game paddles, color monitor optional.
ISBN 0-672-26241-X **$19.95**

☐ VOYAGE OF THE VALKYRIE

Conquer the island of Fugloy and claim its gold. By careful exploration, you must discover the location of 10 castles. Leo Christopherson. Requires Apple II compatible system, 48K RAM, DOS 3.3, Applesoft in ROM, one disk drive, game paddles or joystick.
ISBN 0-672-26238-X **$19.95**

☐ CAVES OF OLYMPUS

A classic adventure set on the planet Olympus. Outwit the Overseer and escape to safety. Noone and Noone. Requires Apple II compatible system, 48K RAM, DOS 3.3, Applesoft in ROM, one disk drive.
ISBN 0-672-26239-8 .... **$19.95**

RESPONSE CARD

☐ Keep me informed of any updates and additions to the Enhancing series.
☐ Please send me a free Don Lancaster software product and book list.
☐ I am also definitely interested in the new "13 day" pre-release program.
☐ What I need right now are ______________

☐ The next enhancements I want to see are

NAME ______________
ADDRESS- ______________
CITY ______________ STATE __ ZIP______
voice phone ______________
data phone ______________

DISKETTES

Please send me:
☐ Enhance Volume 1 Diskette $19.50
☐ Enhance Volume 2 Diskette $19.50
☐ Assembly Cookbook Diskette $19.50
☐ Apple Writer® AWIIe TOOLKIT (8 disk sides)....................$39.50
☐ ProDOS™ Apple Writer® 2.0 TOOLKIT (8 disk sides)....................$39.50
☐ Old fangled animation demo (Meyer)..$ 9.90
☐ Incredible Secret Money Machine ....$ 7.50
☐ Don Lancaster book and software list ...free

☐ Check enclosed for $ ______________
☐ Charge my VISA/MasterCard account

____ — ____ — ____ — ____

Expiration Date ______________

Signature ______________

NAME ______________
ADDRESS ______________
CITY ______________ STATE __ ZIP______
Please, NO purchase orders, COD, cash, Canadian, or foreign.

DISKETTES

Please send me:
☐ Enhance Volume 1 Diskette $19.50
☐ Enhance Volume 2 Diskette $19.50
☐ Assembly Cookbook Diskette $19.50
☐ Apple Writer® AWIIe TOOLKIT (8 disk sides)....................$39.50
☐ ProDOS™ Apple Writer® 2.0 TOOLKIT (8 disk sides)....................$39.50
☐ Old fangled animation demo (Meyer)..$ 9.90
☐ Incredible Secret Money Machine ....$ 7.50
☐ Don Lancaster book and software list ...free

☐ Check enclosed for $ ______________
☐ Charge my VISA/MasterCard account

____ — ____ — ____ — ____

Expiration Date ______________

Signature ______________

NAME ______________
ADDRESS ______________
CITY ______________ STATE __ ZIP______
Please, NO purchase orders, COD, cash, Canadian, or foreign.

FROM

PLACE POSTAGE HERE

SYNERGETICS
746 First Street
Box 809
Thatcher, AZ 85552

FROM

PLACE POSTAGE HERE

SYNERGETICS
746 First Street
Box 809
Thatcher, AZ 85552

FROM

PLACE POSTAGE HERE

SYNERGETICS
746 First Street
Box 809
Thatcher, AZ 85552

SAMS

Enhancing Your Apple® II and IIe
Volume 2

A few simple enhancements can make a big difference in the way your computer operates! Get this book and you'll get some helpful modifications and techniques you can use right away to:

- Microjustify and proportionally space Applewriter IIe to produce hard copy that's almost as good as typeset copy
- Regain control of your machine by picking up an "old monitor" absolute and unconditional reset
- Eliminate the Wolfenstein SS using 6 new and easy-to-build playing aids
- Capture your own source code for custom modification using the "tearing method"
- Use software to create an exact and jitter-free screen lock, and much more!

Don Lancaster's writing style is light and informative. So learn how to get more from your Apple computer! Let Don Lancaster show you the way.

Don Lancaster heads *Synergetics*, a new-age prototyping and consulting firm involved in micro applications and electronic design. He is the well-known author of Sams CMOS and *TTL Cookbooks*. A pioneer in microcomputers, he introduced the first hobbyist integrated-circuit projects, the first sanely priced digital-electronics modules, the first low-cost TVT-1 video display terminal and the first personal computing keyboards. Lancaster's numerous books and articles on personal computing and electronic applications have set new standards for understandable, useful, and exciting technical writing.

Other SAMS books by Don Lancaster include *Active Filter Cookbook, CMOS Cookbook, TTL Cookbook, RTL Cookbook , TVT Cookbook, Cheap Video Cookbook, Son of Cheap Video, The Hexadecimal Chronicles, The Incredible Secret Money Machine* (available only from *Synergetics*), *Don Lancaster's Micro Cookbooks* Volumes 1 and 2 and *Don Lancaster's Assembly Cookbook for the Apple II and IIe*.

Howard W. Sams & Co., Inc.
4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

$17.95/22425

ISBN: 0-672-22425-9